



Professional C++

# C++高级编程

(美) Nicholas A. Solter 著  
Scott J. Kleper

刘鑫 杨健康 等译



机械工业出版社  
China Machine Press

# Professional C++

# C++高级编程

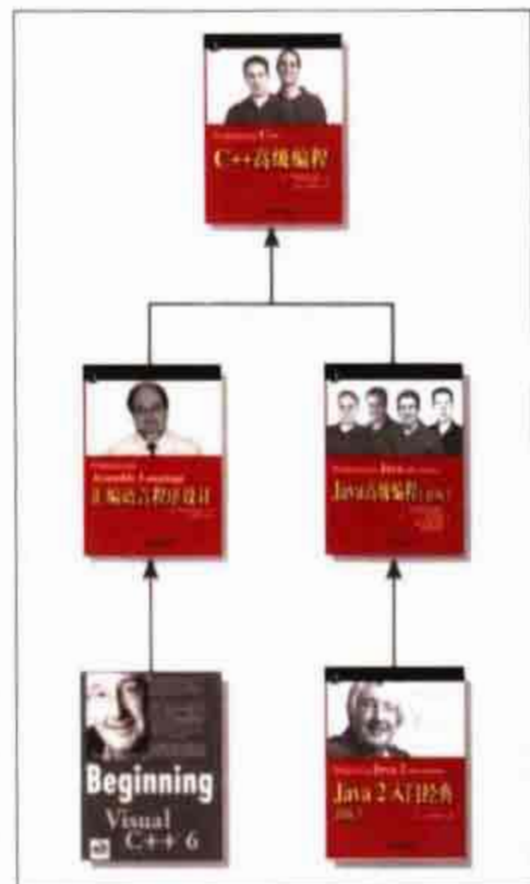
## 本书主要内容:

- 不同的编程方法和高质量的编程风格
- 充分利用C++完成大型软件开发的各种方法
- 确保无错代码的方法
- 认识面向对象设计
- 使用库和模式来提高编程效率、提高编程质量的若干方法
- C++中管理内存的最佳方法
- 输入/输出技术

## 作者简介:

**Nicholas A. Solter** 现就职于Sun Microsystems公司,曾在斯坦福大学攻读计算机专业,获得理学学士和理学硕士学位。他具有丰富的C/C++编程经验和计算机游戏开发经历,并作为助理教授在Fullerton学院讲授过一年C++课程。

**Scott J. Kleper** Reactivity公司的高级软件工程师。在初中就开始了他的编程生涯,用BASIC为Tandy TRS-80编写过一些冒险游戏。在斯坦福大学就读期间,他担任过程序设计入门和面向对象程序设计等多门课程的助教,并获得了计算机科学的理学学士和硕士学位。毕业后致力于人机交互领域,担任过多家公司开发小组的首席工程师。



上架推荐: 计算机/程序设计

ISBN 7-111-17778-9



9 787111 177784



华章图书

华章网站 <http://www.hzbook.com>

网上购书: [www.china-pub.com](http://www.china-pub.com)

投稿热线: (010) 88379604

购书热线: (010) 68995259, 68995264

读者信箱: [hzjsj@hzbook.com](mailto:hzjsj@hzbook.com)

ISBN 7-111-17778-9/TP · 4528

定价: 88.00 元





Professional C++

# C++高级编程

(美) Nicholas A. Solter 著  
Scott J. Kleper

刘鑫 杨健康 等译



机械工业出版社  
China Machine Press

本书既系统全面又突出重点，作者从 C++ 基础知识讲起，始终着眼于 C++ 语言的编程实践，提供了大量实践示例和解决方案，包括如何更好地实现重用、如何有效地测试和调试等 C++ 专业人员常用的一些技术与方法，还提供了一些鲜为人知的、能大大简化工作的 C++ 语言特性；最后，还配有大量可重用的编码模式，并在附录中提供 C++ 面试宝典作为开发人员的实用指南。

本书面向进阶 C++ 的初学者，以及那些想把 C++ 水平提高到专业水准的程序员和开发人员。

Nicholas A. Solter, Scott J. Kleper: Professional C++ (ISBN: 0-7645-7484-1)

Authorized translation from the English language edition published by John Wiley & Sons, Inc.

Copyright ©2005 by Wiley publishing, Inc.

All rights reserved.

本书中文简体字版由约翰-威利父子公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2005-1309

### 图书在版编目 (CIP) 数据

C++ 高级编程 / (美) 索尔特 (Nicholas A. S.), (美) 凯乐普 (Scott J. K.) 著; 刘鑫等译. —北京: 机械工业出版社, 2006. 1

书名原文: Professional C++

ISBN 7-111-17778-9

I. C… II. ①索… ②凯… ③刘… III. C 语言-程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2005) 第 144910 号

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 孙笑竹 姜淑欣

北京瑞德印刷有限公司印刷·新华书店北京发行所发行

2006 年 1 月第 1 版第 1 次印刷

787mm×1092mm 1/16·44 印张

印数: 0 001-4 000 册

定价: 88.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线: (010) 68326294

# 译者序

市场上的 C++ 书籍可谓不少，但面向的读者大多是那些对 C++ 已经很了解的人，还有一些 C++ 书籍更像是参考手册，而不是真正的编程书，并没有真正教我们如何使用 C++。如果读者还不精通 C++，但是想利用它来解决实际问题，这本书就非常适合。它深入浅出地系统介绍了 C++ 的各项高级主题，可以很好地帮助你成为 C++ 专家，而且不要求你先前对 C++ 有太多了解。

本书除了系统、全面的内容介绍外，还讲述了程序设计实践和软件工程，这也是它的另一个闪光点。并不是每个程序员都受过软件工程和软件开发方面的培训，这本书介绍了一些非常好的实践解决方案，告诉我们如何更好地实现重用、如何更快地调试等，这对我们的实际编程尤其有意义。尽早地掌握这些编程实践经验，将有助于编程新手养成良好的编程习惯，即使是具备相当编程经验的人也可以从本书了解到使用 C++ 的更有效方法。

本书有以下特点：

- **重视风格。**如果不注意编程风格，尽管你完全了解 C++，仍有可能写出极糟糕的 C++ 程序，所以这本书中风格问题贯穿始终。
- **突出重点。**本书明确指出了哪些特性很难用、哪些方面很少用。由于 C++ 是一个如此庞大的语言，所以读者要想真正掌握，必须切中要害，强调重点。
- **强调实战。**本书没有太多“玩具型”的小例子，而是提供尽可能多的实践示例，这些示例的代码都可以真正用在你的实际工作中。
- **关注模式。**利用可重用的模式可以编写出更好的代码。本书特别强调了 C++ 程序中反复出现的一些好技术，尤其着很多笔墨来介绍一些可以重用的设计模式。

尽管本书篇幅不短，但是读者读起来一点儿都不会吃力。另外，书中最后还附了一个面试宝典，这是一般的编程书所没有的，这也充分体现出这本书的实用价值。

译者认为，无论本书是作为正式教材还是自学用书，都非常适合。如果你想改进代码质量，提高编程效率，成为一个专业的 C++ 程序员，就千万不要错过这本书。

我们衷心地感谢我们的家人和朋友。在翻译过程中，他们给予了我们莫大的关心、支持和帮助。

全书由刘鑫、杨健康、王林绪、孙健、阎慧、熊伟、朱涛江、王宇、谢剑薇、王树春、韦群、林华君、刘名臣、赵蓓、潘森、刘立强、龚雪晶、王志琳、刘跃邦、蔡洪量、王三梅、何跃强、苏金国、丁小峰、孙春娟、阎文丽、林琪、周兴汉、张练达等进行翻译，其中，刘鑫、杨健康担任主要翻译，王林绪、孙健等进行全书术语的审核，刘名臣、赵蓓等提供技术问题支持，全体工作人员共同完成了本书的翻译工作，最后由刘鑫统稿。

由于时间仓促，且译者的水平有限，在翻译过程中难免会出现一些错误，请读者批评指正。



# 前言

多年以来，在编写速度快、功能强的企业级面向对象程序时，C++已经成为事实上的标准语言。令人惊讶的是，尽管C++变得如此普及，我们却很难全面地掌握这种语言。一些专业C++程序员会使用一些简单但功能很强大的技术，但以往传统的资料中对此都未曾提及；另外，C++中还有一些有用的部分，这些内容即使是对经验丰富的C++程序员来说可能也很神秘。

通常，编程方面的书更多地强调语言的语法，而不注重讲述如何实际使用这种语言来编程。一般的C++书都会分章介绍C++语言的各个主要部分，来解释相关的语法，并提供一个例子。本书不打算落入这种“俗套”。一般的图书只介绍这种语言方方面面的具体细节，而不关注实践内容，本书则不同，我们的目的很明确，就是要教你如何在实际工作中使用C++。你会从书中了解到一些鲜为人知的特性，这些特性能使你的开发更为轻松；另外这里还提供了一些可重用的编码模式，专业的程序员就是因为掌握了这些模式而从初学者中脱颖而出。

## 本书读者对象

即使你用C++已经很多年了，对这种语言的一些更为高级的特性可能还是不太熟悉，或者并没有充分利用到C++的全部功能。也许你编写的C++代码确实也能完成任务，但是你还想更多地了解如何完成C++设计，以及怎样才是好的编程风格。也许你是刚刚接触C++的初学者，想有一个好的起点，希望了解怎样才能“正确地”编写程序。本书将使你的C++水平更上一个台阶，达到专业水准。

因为这本书的目的是让你进阶，从对C++只有基本或初步的了解，转变成一名专业的C++程序员，因此我们假设你对这种语言已经有一定的认识了。第1章相当于一个复习，其中介绍了C++的基本知识，不过仅凭这一章，并不能取代踏踏实实的培训和具体地使用这种语言。即使你刚开始学习C++，但C编程的经验很丰富，阅读第1章应该也够了，你需要的大多数知识都能从中找到。无论如何，你都应当有牢固的编程基础，除了应该对循环、函数和变量等内容了如指掌外，还应该知道如何组织程序的结构，对诸如递归等基本技术应该也不陌生。另外，你应当对散列表和队列等常用的数据结构有一定了解，还应该知道排序和查找等有用的算法。当然，你可以不了解面向对象编程，这部分内容将在第3章介绍。

你可以采用任何编译器来开发代码，但必须熟悉所用的编译器。本书不会提供各种编译器的具体用法说明，你可以参考编译器随附的文档来回顾有关的内容。

## 本书内容

本书提供了一种C++编程方法，这种方法不仅可以改进你的代码质量，还可以提高编程效率。本书不单单讲述C++的语法和语言特性，它还强调了一些编程方法、可重用的设计模式以及好的编程风格。其中，编程方法涵盖了整个软件开发过程，从开始设计和编写代码，到测试、调试和分组工作都有涉及。学完本书，你将掌握C++语言和它的诸多特性，并能充分利用C++的强大功能来完成大规模软件开发。

假设有人已经学过C++的所有语法，但没有见过任何一个简单实例，这就很危险了！没有做过或看

过具体的例子，他可能会认为所有代码都应当放在程序的 `main()` 函数中，或者认为所有变量都应当是全局变量，而通常这些做法都是不好的编程实践。

专业的 C++ 程序员除了了解 C++ 的语法之外，还知道如何正确地使用这种语言。他们认识到好的设计极其重要，并了解面向对象编程理论，知道有哪些最佳的方法来使用现有的库。这些专业的程序员已经开发了大量有用的代码，并提出了许多可重用的思想。

通过阅读本书，你将成为一个专业的 C++ 程序员。你对 C++ 的了解将更为深入，会掌握一些鲜为人知而且通常被误解的语言特性。除此以外，你将学习面向对象程序设计的内容，并获得一些高超的调试技巧。最重要的是，读过这本书后，你的脑海中会留下许多可重用思想，这些思想能够用于实际日常工作当中。

为什么费心尽力地想要成为一个专业的 C++ 程序员，而不是一个只了解 C++ 皮毛的程序员，原因有很多。如果能通晓 C++ 语言的实际工作原理，将大大改善你的代码质量。通过了解不同的编程方法和过程，将有助于你更好地与你的开发小组协作；若能发现可重用的库和常用的设计模式，将有助于提高你的日常工作效率，并避免重蹈覆辙。所有这些，都将使你成为一个更好的程序员和一个更有价值的员工。不过，就算本书没有带给你升迁之喜，多了解一些总不是坏事吧！

## 本书的组织结构

本书包括 6 大部分。

第一部分，“专业 C++ 程序设计概述”，先提供 C++ 基础知识的快速入门课程，为你奠定一定的 C++ 基础。在入门课程之后，将分析 C++ 设计方法。你会了解到设计的重要性、面向对象方法、库和模式的使用、代码重用的重要性，以及当前为众多编程机构所用的工程实践方法。

第二部分，“编写 C++ 代码方式”，这一部分从专业角度为读者提供了一次 C++ 技术之旅。从中可了解到如何编写可读的 C++ 代码，如何创建可重用的类，以及如何充分利用诸如继承和模板等重要的语言特性。

第三部分，“掌握 C++ 高级特性”，在此介绍了如何更充分地利用 C++。本书这一部分展示了 C++ 的诸多神秘之处，并介绍了如何使用这样一些更高级的特性。在这一部分中你将看到 C++ 语言中一些不常用甚至有些古怪的部分，并了解 C++ 中管理内存有哪些好方法，此外还将学习输入输出技术、专业级错误处理、高级的操作符重载、如何编写高效的 C++ 代码，以及如何编写跨语言和跨平台的代码。

第四部分，“确保无错代码”，这一部分的重点是如何编写企业质量的 (enterprise-quality) 软件。你将了解一些软件测试概念，如单元测试和回归测试，还将学习调试 C++ 程序时会用到的一些技术。

第五部分，“使用库和模式”，这一部分介绍了库和模式的使用，基于库和模式的编程，不仅可以使你更省力，还可帮助你编写出更好的代码。你将了解 C++ 提供的标准库，包括诸如扩展标准库的一些高级主题。你还将学习分布式对象、可重用 C++ 设计技术和概念上的面向对象设计模式的有关内容。

本书最后一部分对各章提供了一个实用指南，以方便查阅有关的 C++ 技术。在本书相关网站上 ([www.wrox.com](http://www.wrox.com))，还能找到 C++ 标准库的一个实用参考指南。

## 使用本书的前提

[http://www.wrox.com/WileyCDA/WroxTitle/Professional-C-.productCd-0764574841\\_descCd-DOWNLOAD.html](http://www.wrox.com/WileyCDA/WroxTitle/Professional-C-.productCd-0764574841_descCd-DOWNLOAD.html)

要使用这本书，只要有一个安装了 C++ 编译器的计算机就足够了。不同的编译器在对 C++ 语言的解释上往往存在差别，不过本书只关注 C++ 中已经标准化的部分。本书中的所有程序已经在 Windows、Solaris 和 Linux 等平台上成功地通过了测试。

## 本书约定

为了帮助你更充分地利用这本书，并了解会出现什么情况，我们将采用如下约定：

诸如此类的方框提供了一些不容忘记的重要信息，这些信息与方框前后的文字有很直接的关系。

对当前讨论的主题可能有一些提示、技巧和旁注，这些都将如此缩进并用楷体显示。

正文中还包括以下样式：

- 在初次介绍一些重要的词时，我们会用楷体突出强调。
- 按键采用如下形式表示：Ctrl+A。
- 文件名、URL 和正文中出现的代码表示为：monkey.cpp。
- 代码的表示分为两种：

代码示例中，新出现的代码或重要代码用灰色背景突出强调。

对当前讨论不太重要的代码，或者是前面已经出现过的代码不用灰色背景强调。

## 源代码

在使用本书中的例子时，你可以手工输入所有代码，也可以直接使用本书随附的源代码文件。本书中用到的所有源代码文件都可以从 [www.wrox.com](http://www.wrox.com) 下载。访问该网站时，只要找到本书的书名（可以使用搜索（Search）框，也可以使用某个书目列表），并点击该书详细信息网页上的下载代码（Download Code）链接，就可以得到本书的所有源代码。

由于会有许多书名雷同，最快捷的方法是利用 ISBN 搜索，本书的 ISBN 是 0-7645-7484-1。

下载了代码之后，只需用你最习惯的压缩工具解压即可。另外也可以前往 Wrox 主站的代码下载网页（[www.wrox.com/dynamic/books/download.aspx](http://www.wrox.com/dynamic/books/download.aspx)），在此可以找到这本书以及所有其他 Wrox 书的可用代码。

## 勘误

尽管我们竭尽所能力争文字和代码不出错，但人无完人，错误在所难免。如果你在书中发现了错误（如拼写错误或者某段代码有误），希望你能及时反馈给我们，对此我们表示深深的谢意。你所提供的勘误可能会帮助另一个读者节省大量时间，否则他可能会因为同一处错误迷惑不解，而徒劳地花费数小时想去搞清楚；与此同时，你的勘误还将有助于我们提供更高品质的书。

要找到本书的勘误页面，请访问 [www.wrox.com](http://www.wrox.com)，并使用搜索（Search）框或者某个书目列表找到这本书的书名。然后，在本书的详细信息页面上，单击本书勘误（Book Errata）链接。在这个页面上，你将看到针对这本书提交并由 Wrox 编辑确认发布的所有勘误。在 [www.wrox.com/misc-pages/book-list.shtml](http://www.wrox.com/misc-pages/book-list.shtml) 上还可以得到一个完整书目列表，其中也提供了每本书的勘误。

如果你在本书勘误（Book Errata）页面上没有找到你发现的错误，请访问 [www.wrox.com/contact/techsupport.shtml](http://www.wrox.com/contact/techsupport.shtml)，填写表单，把找到的错误发送给我们。我们将检查你提供的信息，如果错误属实，我们会在本书勘误页面上发布这条信息，并在本书的后续版本中修正这个问题。

## p2p.wrox.com

要与作者或其他人讨论有关 C++ 技术问题，请加入 P2P 论坛（[p2p.wrox.com](http://p2p.wrox.com)）。这个论坛是一个基



于 Web 的系统，你可以在此发表有关 Wrox 图书和相关技术的消息，并与其他读者和技术用户交流。论坛提供了一个订购功能，针对你感兴趣的主题，如果论坛上新发布了相关的消息，会通过电子邮件通知你。Wrox 作者、编辑、其他行业专家以及其他读者也会访问这些论坛。

在 <http://p2p.wrox.com> 上，你会看到许多论坛，这些论坛不仅可以帮助你阅读本书，还有助于你开发自己的应用软件。要想加入论坛，只需遵循以下几个步骤：

1. 访问 [p2p.wrox.com](http://p2p.wrox.com)，并单击“注册”（Register）链接。
2. 阅读使用条文，并单击“同意”（Agree）。
3. 填写加入论坛的必要信息，如果想提供其他可选信息，也可以相应填写，单击“提交”（Submit）。
4. 你将收到一个电子邮件，其中将说明如何验证你的账户，并完成加入过程。

如果你只是阅读论坛中的消息，无须加入 P2P，不过，如果你想发布自己的消息，就必须加入论坛。

一旦加入，你就可以发布新的消息了，还可以对其他用户发布的消息做出响应。任何时刻你都可以在 Web 上阅读消息。如果你希望某个论坛能通过电子邮件向你发送新发布的消息，请点击论坛列表中该论坛名旁边的“订购此论坛”（Subscribe to this Forum）图标。

要了解如何使用 Wrox P2P 的更多信息，请阅读 P2P FAQs，在此对这个论坛软件工作的问题做了解释，另外还回答了与 P2P 和 Wrox 书有关的许多常用问题。读者若要阅读 FAQs，单击任何 P2P 页面上的 FAQ 链接都可以。

## 致谢

本书能成功问世，离不开许多人的帮助，为此我们也欠下了很多人情。这里要感谢 Waterside Productions 的 David Fugate 对我们的谆谆教导，还要感谢 Wiley 的 Robert Elliot，是他让一文不名的我们有机会写出这本书，以一种全新的方式讲述 C++。如果没有策划编辑 Adaobi Obi Tulton 的大力协助，这本书的出版可能不会像现在这么好。另外要感谢 Kathryn Malm Bourgoine 在编辑方面提供的帮助。封面上的照片巧妙地对我们有所美化，感谢 Adam Tow 拍摄的这张照片。

还要感谢我们的所有同事和老师，是诸位多年以来一直在鼓励我们以正确的方法编写代码。特别地，我们要感谢 Mike Hanson、Maggie Johnson、Adam Nash、Nick Parlante、Bob Plummer、Eric Roberts、Mehran Sahami、Bill Walker、Dan Walkowski、Patrick Young 和 Julie Zelenski。还要向 Jerry Cain 致以诚挚的谢意，是他最早教我们学 C++，不仅如此，他还担任了本书的技术编辑，一丝不苟地分析了本书中的代码，认真的程度就好像在批阅我们的期末试卷一样。

另外要感谢：Rob Baesman、Aaron Bradley、Elaine Cheung、Marni Kleper、Toli Kuznets、Akshay Rangnekar、Eltefaat Shokri、Aletha Solter、Ken Solter 和 Sonja Solter，各位都审阅了书中的一章甚至多章。当然，如果书中还存在错误，这都要归咎于我们自己。在此还要向各位的家人表示感谢，谢谢大家的耐心和支持。

最后，我们还要感谢你，亲爱的读者，谢谢你打算采用我们提供的方法踏上专业 C++ 开发的道路。

# 作者简介

**Nicholas A. Solter** 曾在斯坦福大学攻读计算机专业，获得了理学学士和理学硕士学位，并致力于多个系统的开发。他还是一个学生时，就担任了多门计算机课程的助教，这些课程门类众多，下至为非计算机专业开设的入门性课程，上至有关项目组织和软件工程的高级课程。

Nick 现在是一名软件工程师，任职于 Sun Microsystems 公司，在工作中他主要用 C 和 C++ 编程来开发高可用性软件。他还在计算机游戏行业有过工作经历。在 Digital Media International 公司工作期间，他曾担任多媒体教学游戏“Land Before Time Math Adventure”的首席程序员。在 Electronic Arts 公司实习时，他协助开发了 Course Architect 2000 高尔夫教程，高尔夫界大名鼎鼎的“老虎”伍兹参加 PGA（美国职业高尔夫球协会）Tour 2000 比赛时就使用了这个工具。

除了行业开发经验外，Nick 还作为兼职教授在 Fullerton 学院讲授了一年 C++ 课程。闲暇时，Nick 很喜欢看书、玩篮球、照顾他的儿子 Kai，以及和家人在一起享受天伦之乐。

**Scott J. Kleper** 在初中时就开始了他的编程生涯，那时他用 BASIC 为 Tandy TRS-80 编写了一些冒险游戏。由于他所在的高中 Mac 大行其道，Scott 也因此开始转向更高级的语言，并发布过许多荣获大奖的共享应用软件。

Scott 也考入了斯坦福大学，在那里他获得了计算机科学专业的理学学士和理学硕士学位，并致力于人机交互领域。上大学期间，Scott 担任了一些课程的助教，包括程序设计入门、面向对象程序设计、数据结构、GUI 框架、项目组织和 Internet 编程等。

毕业以后，Scott 担任了多家公司开发小组的首席工程师，目前是 Reactivity 公司的一名高级软件工程师。工作之余，Scott 特别喜欢网上购物和读书，他还是一个不错的吉它手。

## 特别鸣谢

副总裁、主管集团出版商

Richard Swadley

副总裁、出版商

Joseph B. Wikert

执行编辑

Robert Elliott

编辑经理

Kathryn Malm Bourgoine

高级制作编辑

Geraldine Fahey

高级策划编辑

Adaobi Obi Tulton

制作编辑

Felicia Robinson

媒体开发专家

Richard Graves

技术编辑

Jerry Cain

文字设计与合成

Wiley Composition Services

封面摄影

Adam Tow

# 目 录

译者序

前言

作者简介

## 第一部分 专业 C++ 程序设计概述

第 1 章 C++ 快速入门 .....	1
1.1 C++ 基础 .....	1
1.1.1 必不可少的“Hello, World” .....	1
1.1.2 命名空间 .....	4
1.1.3 变量 .....	5
1.1.4 操作符 .....	7
1.1.5 类型 .....	9
1.1.6 条件语句 .....	10
1.1.7 循环 .....	13
1.1.8 数组 .....	14
1.1.9 函数 .....	14
1.1.10 结束语 .....	15
1.2 C++ 进阶 .....	16
1.2.1 指针和动态内存 .....	16
1.2.2 C++ 中的字符串 .....	18
1.2.3 引用 .....	20
1.2.4 异常 .....	21
1.2.5 const 的多重用途 .....	22
1.3 作为一种面向对象语言的 C++ .....	23
1.4 你的第一个实用的 C++ 程序 .....	25
1.4.1 一个员工记录系统 .....	26
1.4.2 Employee 类 .....	26
1.4.3 Database 类 .....	30
1.4.4 用户界面 .....	34
1.4.5 对程序的评价 .....	36
1.5 小结 .....	36
第 2 章 设计专业的 C++ 程序 .....	37

2.1 什么是编程设计 .....	37
2.2 编程设计的重要性 .....	38
2.3 C++ 设计有什么不同之处 .....	39
2.4 C++ 设计的两个原则 .....	40
2.4.1 抽象 .....	40
2.4.2 重用 .....	41
2.5 设计一个象棋程序 .....	43
2.5.1 需求 .....	43
2.5.2 设计步骤 .....	43
2.6 小结 .....	47
第 3 章 基于对象的设计 .....	48
3.1 面向对象的世界观 .....	48
3.1.1 我是在以过程性思维思考吗 .....	48
3.1.2 面向对象思想 .....	49
3.1.3 身处对象世界中 .....	51
3.1.4 对象关系 .....	52
3.1.5 抽象 .....	61
3.2 小结 .....	63
第 4 章 基于库和模式的设计 .....	64
4.1 重用代码 .....	64
4.1.1 有关术语 .....	64
4.1.2 决定是否重用代码 .....	65
4.1.3 重用代码的策略 .....	67
4.1.4 捆绑第三方应用 .....	70
4.1.5 开源库 .....	70
4.1.6 C++ 标准库 .....	71
4.2 利用模式和技术完成设计 .....	81
4.2.1 设计技术 .....	81
4.2.2 设计模式 .....	82
4.3 小结 .....	83
第 5 章 重用设计 .....	84
5.1 重用方法论 .....	84



5.2 如何设计可重用的代码 .....	85	7.5.1 使用常量 .....	119
5.2.1 使用抽象 .....	85	7.5.2 利用 const 变量 .....	120
5.2.2 适当地建立代码结构以优化重用 .....	86	7.5.3 使用引用而不是指针 .....	120
5.2.3 设计可用的接口 .....	89	7.5.4 使用定制异常 .....	120
5.2.4 协调一般性和易用性 .....	93	7.6 格式化 .....	121
5.3 小结 .....	94	7.6.1 有关大括号对齐的争论 .....	121
第6章 充分利用软件工程方法 .....	95	7.6.2 考虑空格和小括号 .....	122
6.1 为什么需要过程 .....	95	7.6.3 空格和制表符 .....	122
6.2 软件生命期模型 .....	96	7.7 风格方面的难题 .....	122
6.2.1 分阶段模型和瀑布模型 .....	96	7.8 小结 .....	123
6.2.2 螺旋方法 .....	98	第8章 掌握类和对象 .....	124
6.2.3 统一开发过程 .....	100	8.1 电子表格示例 .....	124
6.3 软件工程方法论 .....	101	8.2 编写类 .....	124
6.3.1 极限编程(XP) .....	101	8.2.1 类定义 .....	124
6.3.2 软件 triage .....	104	8.2.2 定义方法 .....	127
6.4 建立自己的过程和方法论 .....	104	8.2.3 使用对象 .....	130
6.4.1 以开放的心态接纳新思想 .....	105	8.3 对象生命期 .....	131
6.4.2 汇总新思想 .....	105	8.3.1 对象创建 .....	131
6.4.3 明确哪些可行, 哪些不可行 .....	105	8.3.2 对象撤销 .....	140
6.4.4 不要做叛逃者 .....	105	8.3.3 对象赋值 .....	141
6.5 小结 .....	105	8.3.4 区别复制和赋值 .....	143
第二部分 编写 C++ 代码方式		8.4 小结 .....	144
第7章 好的编码风格 .....	107	第9章 精通类和对象 .....	146
7.1 为什么代码看上去要好 .....	107	9.1 对象中的动态内存分配 .....	146
7.1.1 提前考虑 .....	107	9.1.1 Spreadsheet 类 .....	146
7.1.2 保持清晰 .....	107	9.1.2 用析构函数释放内存 .....	147
7.1.3 好的代码风格包括哪些因素 .....	108	9.1.3 处理复制和赋值 .....	147
7.2 为代码加注释 .....	108	9.2 不同类型的数据成员 .....	154
7.2.1 写注释的原因 .....	108	9.2.1 静态数据成员 .....	154
7.2.2 注释风格 .....	111	9.2.2 const 数据成员 .....	156
7.2.3 本书中的注释 .....	115	9.2.3 引用数据成员 .....	157
7.3 分解 .....	115	9.2.4 const 引用数据成员 .....	158
7.3.1 通过重构来分解 .....	115	9.3 深入了解方法 .....	158
7.3.2 根据设计来分解 .....	116	9.3.1 静态方法 .....	158
7.3.3 本书中的分解 .....	117	9.3.2 const 方法 .....	159
7.4 命名 .....	117	9.3.3 方法重载 .....	161
7.4.1 选择一个好名字 .....	117	9.3.4 默认参数 .....	162
7.4.2 命名约定 .....	118	9.3.5 内联方法 .....	163
7.5 合理地使用语言特性 .....	119	9.4 嵌套类 .....	164
		9.5 友元 .....	166
		9.6 操作符重载 .....	166

9.6.1 实现加法 .....	166	11.1 模板概述 .....	219
9.6.2 重载算术操作符 .....	170	11.2 类模板 .....	220
9.6.3 重载比较操作符 .....	172	11.2.1 编写类模板 .....	220
9.6.4 利用操作符重载构建类型 .....	174	11.2.2 编译器如何处理模板 .....	227
9.7 方法和成员指针 .....	174	11.2.3 模板代码在文件之间的分布 .....	228
9.8 构建抽象类 .....	175	11.2.4 模板参数 .....	229
9.9 小结 .....	178	11.2.5 方法模板 .....	231
第 10 章 探索继承技术 .....	179	11.2.6 模板类特殊化 .....	235
10.1 使用继承构建类 .....	179	11.2.7 从模板类派生子类 .....	239
10.1.1 扩展类 .....	179	11.2.8 继承与特殊化的区别 .....	240
10.1.2 覆盖方法 .....	182	11.3 函数模板 .....	240
10.2 继承以实现重用 .....	184	11.3.1 函数模板特殊化 .....	241
10.2.1 类 WeatherPrediction .....	184	11.3.2 函数模板的重载 .....	242
10.2.2 在子类中增加功能 .....	185	11.3.3 类模板的友元函数模板 .....	243
10.2.3 在子类中进行功能替换 .....	187	11.4 高级模板 .....	244
10.3 考虑父类 .....	187	11.4.1 关于模板参数的更多知识 .....	244
10.3.1 父构造函数 .....	187	11.4.2 模板类的部分特殊化 .....	251
10.3.2 父析构函数 .....	189	11.4.3 用重载模板函数部分特殊化 .....	256
10.3.3 引用父类的数据 .....	191	11.4.4 模板递归 .....	257
10.3.4 向上类型强制转换和向下类型 强制转换 .....	192	11.5 小结 .....	264
10.4 继承以实现多态 .....	193	第 12 章 理解 C++ 疑难问题 .....	265
10.4.1 Spreadsheet 的返回结果 .....	193	12.1 引用 .....	265
10.4.2 设计多态电子表格单元格 .....	194	12.1.1 引用变量 .....	265
10.4.3 电子表格单元格的基类 .....	194	12.1.2 引用数据成员 .....	267
10.4.4 各个子类 .....	196	12.1.3 引用参数 .....	267
10.4.5 充分利用多态 .....	198	12.1.4 引用返回类型 .....	268
10.4.6 将来的考虑 .....	199	12.1.5 采用引用还是指针 .....	268
10.5 多重继承 .....	200	12.2 关键字疑点 .....	270
10.5.1 从多个类继承 .....	200	12.2.1 关键字 const .....	270
10.5.2 命名冲突与二义基类 .....	201	12.2.2 关键字 static .....	273
10.6 继承技术中有趣而隐蔽的问题 .....	203	12.2.3 非局部变量的初始化顺序 .....	276
10.6.1 改变覆盖方法的特性 .....	203	12.3 类型与类型强制转换 .....	276
10.6.2 覆盖方法的特殊情况 .....	206	12.3.1 typedef .....	276
10.6.3 复制构造函数与相等操作符 .....	212	12.3.2 类型强制转换 .....	277
10.6.4 关键字 virtual 的真相 .....	213	12.4 解析作用域 .....	281
10.6.5 运行时类型工具 .....	215	12.5 头文件 .....	282
10.6.6 非公共继承 .....	217	12.6 C 实用工具 .....	283
10.6.7 虚基类 .....	217	12.6.1 变量长度参数列表 .....	283
10.7 小结 .....	218	12.6.2 预处理宏 .....	285
第 11 章 利用模板编写通用代码 .....	219	12.7 小结 .....	285

### 第三部分 掌握 C++ 高级特性

第 13 章 有效的内存管理 .....	287	14.5.1 宽字符 .....	329
13.1 使用动态内存 .....	287	14.5.2 非西方字符集 .....	329
13.1.1 如何描述内存 .....	287	14.5.3 本地化环境与方面 .....	330
13.1.2 内存的分配与撤销 .....	289	14.6 小结 .....	332
13.1.3 数组 .....	291	第 15 章 处理错误 .....	333
13.1.4 使用指针 .....	297	15.1 错误和异常 .....	333
13.2 数组与指针的对应 .....	299	15.1.1 到底什么是异常 .....	333
13.2.1 数组即指针 .....	299	15.1.2 C++ 中的异常为什么好 .....	334
13.2.2 指针并非都是数组 .....	300	15.1.3 C++ 中的异常为什么不好 .....	335
13.3 动态字符串 .....	300	15.1.4 我们的建议 .....	335
13.3.1 C 风格的字符串 .....	301	15.2 异常机制 .....	335
13.3.2 字符串直接量 .....	302	15.2.1 抛出和捕获异常 .....	336
13.3.3 C++ 的字符串类 .....	303	15.2.2 异常类型 .....	338
13.4 低级的内存操作 .....	304	15.2.3 抛出和捕获多个异常 .....	339
13.4.1 指针运算 .....	304	15.2.4 未捕获的异常 .....	342
13.4.2 自定义内存管理 .....	305	15.2.5 抛出列表 .....	343
13.4.3 垃圾回收 .....	305	15.3 异常和多态 .....	346
13.4.4 对象池 .....	306	15.3.1 标准异常层次体系 .....	346
13.4.5 函数指针 .....	306	15.3.2 按类层次捕获异常 .....	348
13.5 常见的内存陷阱 .....	308	15.3.3 编写自己的异常类 .....	349
13.5.1 字符串空间分配不足 .....	308	15.4 栈展开和清除 .....	351
13.5.2 内存泄漏 .....	309	15.4.1 捕获、清除和重新抛出 .....	353
13.5.3 二次删除与无效指针 .....	311	15.4.2 使用智能指针 .....	353
13.5.4 访问越界指针 .....	312	15.5 常见的错误处理问题 .....	354
13.6 小结 .....	312	15.5.1 内存分配错误 .....	354
第 14 章 揭开 C++ I/O 的神秘		15.5.2 构造函数中的错误 .....	356
面纱 .....	313	15.5.3 析构函数中的错误 .....	357
14.1 使用流 .....	313	15.6 综合 .....	357
14.1.1 到底什么是流 .....	313	15.7 小结 .....	359
14.1.2 流的源与目的 .....	313	第四部分 确保无错代码	
14.1.3 流输出 .....	314	第 16 章 重载 C++ 操作符 .....	361
14.1.4 流输入 .....	317	16.1 操作符重载概述 .....	361
14.1.5 输入与输出对象 .....	321	16.1.1 为什么要重载操作符 .....	362
14.2 字符串流 .....	323	16.1.2 操作符重载的限制 .....	362
14.3 文件流 .....	324	16.1.3 操作符重载中的选择 .....	362
14.3.1 使用 seek() 与 tell() .....	325	16.1.4 不应重载的操作符 .....	364
14.3.2 链接流 .....	327	16.1.5 可重载操作符小结 .....	364
14.4 双向 I/O .....	327	16.2 重载算术操作符 .....	367
14.5 国际化 .....	329	16.2.1 重载一元减和一元加 .....	367
		16.2.2 重载自增和自减 .....	368



16.3 重载位操作符和二元逻辑操作符 .....	369	18.1.2 实现问题 .....	413
16.4 重载插入和析取操作符 .....	369	18.1.3 特定于平台的特性 .....	414
16.5 重载下标操作符 .....	371	18.2 跨语言开发 .....	415
16.5.1 利用 operator[] 提供只读 访问 .....	373	18.2.1 混合 C 和 C++ .....	415
16.5.2 非整数数组索引 .....	375	18.2.2 转换模式 .....	415
16.6 重载函数调用操作符 .....	375	18.2.3 与 C 代码的链接 .....	418
16.7 重载解除引用操作符 .....	377	18.2.4 利用 JNI 混合 Java 和 C++ .....	419
16.7.1 实现 operator* .....	378	18.2.5 C++ 与 Perl 和 Shell 脚本的 混合 .....	421
16.7.2 实现 operator-> .....	379	18.2.6 C++ 与汇编代码的混合 .....	424
16.7.3 到底什么是 operator->* .....	379	18.3 小结 .....	424
16.8 编写转换操作符 .....	380	第 19 章 熟练地测试 .....	425
16.8.1 转换操作符的二义性问题 .....	381	19.1 质量控制 .....	425
16.8.2 布尔表达式的转换 .....	382	19.1.1 谁来负责测试 .....	425
16.9 重载内存分配和撤销操作符 .....	383	19.1.2 bug 的生命期 .....	425
16.9.1 new 和 delete 究竟如何工作 .....	384	19.1.3 bug 跟踪工具 .....	427
16.9.2 重载 operator new 和 operator delete .....	385	19.2 单元测试 .....	428
16.9.3 重载带额外参数的 operator new 和 operator delete .....	387	19.2.1 单元测试的方法 .....	428
16.10 小结 .....	389	19.2.2 单元测试过程 .....	429
第 17 章 编写高效的 C++ 程序 .....	390	19.2.3 实战单元测试 .....	431
17.1 性能和效率概述 .....	390	19.3 高级测试 .....	439
17.1.1 实现高效的两种方法 .....	390	19.3.1 集成测试 .....	439
17.1.2 两类程序 .....	390	19.3.2 系统测试 .....	440
17.1.3 C++ 是一种低效语言吗 .....	391	19.3.3 回归测试 .....	441
17.2 语言级效率 .....	391	19.4 成功测试的提示 .....	441
17.2.1 高效地处理对象 .....	392	19.5 小结 .....	442
17.2.2 不要过度使用高开销的语言 特性 .....	394	第 20 章 征服调试 .....	443
17.2.3 使用内联方法和函数 .....	395	20.1 调试基本法则 .....	443
17.3 设计级效率 .....	396	20.2 bug 分类 .....	443
17.3.1 尽可能缓存 .....	396	20.3 避免 bug .....	443
17.3.2 使用对象池 .....	397	20.4 找出 bug 的方法 .....	444
17.3.3 使用线程池 .....	402	20.4.1 错误日志 .....	444
17.4 测评分析 .....	402	20.4.2 调试轨迹 .....	445
17.5 小结 .....	410	20.4.3 断言 .....	455
第 18 章 开发跨平台和跨语言的应用 ..	411	20.5 调试技术 .....	456
18.1 跨平台开发 .....	411	20.5.1 再生 bug .....	456
18.1.1 体系结构问题 .....	411	20.5.2 调试可再生 bug .....	456
		20.5.3 调试不可再生 bug .....	457
		20.5.4 调试内存问题 .....	457
		20.5.5 调试多线程程序 .....	461

20.5.6 调试示例：文章引用 .....	461	22.2.5 编写自己的函数对象 .....	534
20.5.7 从 ArticleCitations 示例学到的 教训 .....	472	22.3 算法细节 .....	535
20.6 小结 .....	472	22.3.1 工具算法 .....	536
第 21 章 深入 STL：容器和迭代器 .....	473	22.3.2 非修改算法 .....	536
21.1 容器概述 .....	473	22.3.3 修改算法 .....	542
21.1.1 元素需求 .....	474	22.3.4 排序算法 .....	545
21.1.2 异常和错误检查 .....	474	22.3.5 集合算法 .....	547
21.1.3 迭代器 .....	475	22.4 算法和函数对象示例：选民注册 审计 .....	549
21.2 顺序容器 .....	476	22.4.1 选民注册审计问题描述 .....	549
21.2.1 vector .....	476	22.4.2 auditVoterRolls() 函数 .....	549
21.2.2 vector<bool> 特殊化 .....	492	22.4.3 getDuplicates() 函数 .....	550
21.2.3 deque .....	493	22.4.4 RemoveNames 函数对象 .....	551
21.2.4 list .....	493	22.4.5 NameInList 函数对象 .....	552
21.3 容器适配器 .....	496	22.4.6 测试 auditVoterRolls() 函数 .....	553
21.3.1 queue .....	496	22.5 小结 .....	554
21.3.2 priority_queue .....	499	第 23 章 定制和扩展 STL .....	555
21.3.3 stack .....	502	23.1 分配器 .....	555
21.4 关联容器 .....	503	23.2 迭代器适配器 .....	556
21.4.1 pair 工具类 .....	503	23.2.1 逆序迭代器 .....	556
21.4.2 map .....	504	23.2.2 流迭代器 .....	557
21.4.3 multimap .....	511	23.2.3 插入迭代器 .....	557
21.4.4 set .....	515	23.3 扩展 STL .....	559
21.4.5 multiset .....	517	23.3.1 为什么要扩展 STL .....	559
21.5 其他容器 .....	517	23.3.2 编写 STL 算法 .....	559
21.5.1 数组作为 STL 容器 .....	517	23.3.3 编写一个 STL 容器 .....	561
21.5.2 string 作为 STL 容器 .....	518	23.4 小结 .....	587
21.5.3 流作为 STL 容器 .....	519	第 24 章 探讨分布式对象 .....	588
21.5.4 bitset .....	519	24.1 分布式计算的魔力 .....	588
21.6 小结 .....	524	24.1.1 分布以获得可扩展性 .....	588
第 5 部分 使用库和模式		24.1.2 分布以获得可靠性 .....	589
第 22 章 掌握 STL 算法和函数对象 .....	525	24.1.3 分布以获得集中性 .....	589
22.1 算法概述 .....	525	24.1.4 分布式内容 .....	589
22.1.1 find() 和 find_if() 算法 .....	526	24.1.5 分布式 vs 网络式 .....	589
22.1.2 accumulate() 算法 .....	528	24.2 分布式对象 .....	590
22.2 函数对象 .....	529	24.2.1 串行化和编组 .....	590
22.2.1 算术函数对象 .....	529	24.2.2 远程过程调用 .....	593
22.2.2 比较函数对象 .....	530	24.3 CORBA .....	595
22.2.3 逻辑函数对象 .....	531	24.3.1 接口定义语言 .....	595
22.2.4 函数对象适配器 .....	531	24.3.2 实现类 .....	597
		24.3.3 使用对象 .....	598

24.4 XML .....	602	26.1.3 使用单例 .....	645
24.4.1 XML 快速入门 .....	602	26.2 工厂模式 .....	646
24.4.2 XML 作为一种分布式对象 技术 .....	604	26.2.1 举例：汽车工厂模拟 .....	646
24.4.3 用 C++ 生成和解析 XML .....	604	26.2.2 实现工厂 .....	648
24.4.4 XML 验证 .....	612	26.2.3 使用工厂 .....	650
24.4.5 用 XML 构建分布式对象 .....	613	26.2.4 工厂的其他使用 .....	651
24.4.6 SOAP (简单对象访问 协议) .....	616	26.3 代理模式 .....	652
24.5 小结 .....	618	26.3.1 举例：隐藏网络连通性问题 .....	652
第 25 章 结合技术和框架 .....	619	26.3.2 实现代理 .....	652
25.1 “我想不起来如何...” .....	619	26.3.3 使用代理 .....	653
25.1.1 .....编写一个类 .....	619	26.4 适配器模式 .....	653
25.1.2 .....派生一个现有类 .....	621	26.4.1 举例：适配一个 XML 库 .....	653
25.1.3 .....抛出和捕获异常 .....	622	26.4.2 适配器的实现 .....	654
25.1.4 .....读文件 .....	622	26.4.3 使用适配器 .....	657
25.1.5 .....写文件 .....	623	26.5 装饰器模式 .....	658
25.1.6 .....编写模板类 .....	623	26.5.1 举例：定义网页中的样式 .....	658
25.2 还有更好的办法 .....	625	26.5.2 装饰器的实现 .....	659
25.2.1 带引用计数的智能指针 .....	625	26.5.3 使用装饰器 .....	660
25.2.2 双重分派 .....	630	26.6 职责链模式 .....	661
25.2.3 混合类 .....	635	26.6.1 举例：事件处理 .....	661
25.3 面向对象框架 .....	637	26.6.2 职责链的实现 .....	661
25.3.1 使用框架 .....	637	26.6.3 使用职责链 .....	662
25.3.2 模型-视图-控制器模式 .....	638	26.7 观察者模式 .....	662
25.4 小结 .....	639	26.7.1 举例：事件处理 .....	663
第 26 章 应用设计模式 .....	640	26.7.2 实现观察者 .....	663
26.1 单例模式 .....	640	26.7.3 使用观察者 .....	664
26.1.1 举例：日志机制 .....	640	26.8 小结 .....	665
26.1.2 单例的实现 .....	641		
		<b>附 录</b>	
		附录 A C++ 面试宝典 .....	666
		附录 B 参考书目 .....	681

# 第一部分

## 专业 C++ 程序设计概述

### 第1章 C++ 快速入门

本章的目标很明确，即简要地介绍 C++ 中最重要的一些环节，使读者对 C++ 有一个基本的了解，以便更好地学习书中余下的内容。本章可不是详尽介绍 C++ 程序设计语言的全面教程，例如“程序是什么”、“= 和 == 之间有什么区别”之类的基本问题这里并不涉及。对于一些深奥的问题，比如“什么是 union?”、“volatile 关键字是怎么回事?”，我们也不打算深入探究。C 语言里的某些部分与 C++ 的关系并不大，另外 C++ 的某些部分会在本书的后续章节做深入的介绍，所以这些内容在本章中也不会出现。

本章就是要介绍每天与程序员“打交道”的基本 C++ 知识。如果你好久没有用 C++ 了，忘记了 for 循环的语法，那么可以在这一章中找回“记忆”。如果你是头一次接触 C++，还不知道什么是引用变量，也可以在这里了解到有关内容。

如果你很熟悉 C++，而且已经有相当丰富的经验了，可以快速地浏览一下本章，看看其中哪些关于 C++ 的基本知识是以前没有注意到、而需要掌握的。如果你是一个 C++ 新手，请花些时间仔细阅读本章，一定要完全掌握本章中的所有例子。如果你还需要更多的相关信息，可以参阅附录 B 中列出的参考书目。

#### 1.1 C++ 基础

C++ 语言通常被视作一种“更好的 C”，或者是“C 的超集”。C 语言中许多麻烦的地方、含糊不清的地方在设计 C++ 时得到了解决。由于 C++ 是基于 C 的，如果你是一个熟练的 C 程序员，就会发现这一节中看到的许多语法都似曾相识。不过，这两种语言当然也存在着差别。C++ 之父 Bjarne Stroustrup 所著的《*The C++ Programming Language*》厚达 911 页，而 Kernighan 和 Ritchie 编写的《*The C Programming Language*》只有区区 274 页，由此就可见一斑。所以，如果你是一位 C 程序员，就要当心你没见过的、不熟悉的语法！

##### 1.1.1 必不可少的“Hello, World”

可以说，下面这段代码可能是你遇到的最简单的 C++ 程序了。（译者注：介绍语言的书几乎都从“Hello, World”开始举例，所以这一节标题为必不可少的“Hello, World”。）

```
// helloworld.cpp

#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

不出所料，这段代码会在屏幕上打印出消息“Hello, World! ”。这是一个极其简单的程序，没有多大的价值，但是由此确实可以体现出 C++ 程序格式方面的一些重要概念。

#### 注释

这个程序的第一行代码是一条注释 (comment)，所谓注释就是仅由程序员阅读的消息，而编译器会将其忽略。C++ 中，注释有两种表示方法。在上述例子中，行首的两个斜线表示当前行上其后的所有内容都是注释。

```
// helloworld.cpp
```

如果使用 C 风格的注释，也可以起到同样的作用（也就是说，仅作为注释，不做任何实际动作），C 风格的注释在 C++ 中也是合法的。C 风格注释以 /\* 开头，以 \*/ 结束。采用这种形式，C 风格的注释可以跨多行，即可以包含多行注释。以下代码显示了一个实际的 C 风格注释（或者，更确切地说，这是一个没有任何实际作为的注释）。

```
/* this is a multiline
 * C-style comment. The
 * compiler will ignore
 * it.
 */
```

第 7 章将详细介绍注释。

#### 预编译器指令

要构建一个 C++ 程序，这个过程可分为三步。首先，代码通过一个预处理器 (preprocessor) 运行，这个预处理器会识别出代码的有关元信息。其次，代码经过编译 (compile)，或翻译为机器可读的对象文件。最后，单个的对象文件链接 (link) 到一起，构成一个应用。交由预处理器处理的指令以 # 字符打头，如前例中的 #include<iostream> 代码行便是如此。在这个例子中，包含 (include) 指令指出，预处理器要取得 iostream 头文件中的所有内容，并使之对当前文件可用。头文件最常见的用法就是声明一些函数，而这些函数在别处定义。要记住，声明 (declaration) 只是告诉编译器如何调用一个函数，定义 (definition) 则包含了函数的实际代码。iostream 头文件声明了 C++ 提供的输入和输出机制。尽管上述程序只有输出文本这样一个简单的任务，但如果它没有包含这个头文件，那么连这个简单任务也无法完成。

在 C 中，所包含的文件通常以 .h 结尾，如 <stdio.h>。在 C++ 中，标准库头文件的后缀可以忽略不要，如 <iostream>。C 中常用的一些标准头文件在 C++ 中仍然存在，不过文件名可能有所不同。例如，通过包含 <cstdio>，可以访问 <stdio.h> 中的功能。

表 1-1 显示了一些最常用的预编译器指令。

表 1-1

预编译器指令	功 能	常见用途
#include [file]	指定的文件 (file) 将插入到指令所在位置代码处	通常用于包含头文件, 使代码可以利用在别处定义的功能
#define [key] [value]	指定键 (key) 的每次出现都要替换为指定的值 (value)	在 C 中, 这通常用于定义一个常量值或一个宏。C++ 则提供了一个定义常量的更好的机制。宏往往很危险, 因此在 C++ 中很少使用 #define, 有关详细内容请见第 12 章
#ifdef [key] #ifndef [key] #endif	这个指令会有条件地包含 (或忽略) ifdef (“如果定义了”) 或 ifndef (“如果没有定义”) 块中的代码, 这取决于是否用 #define 定义了指定值	常用于避免循环包含。所包含的每个文件在最前面定义一个值, 并将余下的代码包围在一个 #ifndef 和 #endif 中, 这样就不会被多次包含
#pragma	这个指令的功能依编译器不同而有所不同。如果在预处理过程中遇到这条指令, 通常允许程序员显示一条警告或错误消息	对于不同的编译器, 由于 #pragma 的用法并不是标准的, 所以我们建议不要使用这条指令

### main 函数

顾名思义, main() 自然就是程序的入口。上例中的 main() 会返回一个 int, 它将指示程序的结果状态。main() 带两个参数: argc 和 argv。argc 给出传递给程序的实参个数, argv 则具体包含这些实参。需要注意, 第一个实参往往就是程序本身的名字。

### I/O 流

如果你刚开始接触 C++, 并且有 C 的背景, 很可能不清楚 std::cout 是什么, 不明白为什么这里不用原先值得信赖的 printf()。尽管在 C++ 用 printf() 也是可以的, 但是 C++ 流库提供的输入/输出工具相比之下要好得多。

我们将在第 14 章深入讨论 I/O 流, 不过输出的基本思想相当简单。可以把输出流看作是数据的一个清洗槽。你扔到这个“槽”里的所有东西都会得到适当的输出。std::cout 就是对应用户控制台 (即标准输出, standard out) 的“槽”。除此之外, 还有其他的一些槽, 如 std::cerr, 它会输出至错误控制台。“<<”操作符就是把数据扔到槽里。在上述例子中, 则是把一个加引号的文本串发送至标准输出。基于输出流, 允许在一行代码上向流中顺序地发送多种不同类型的数据。以下代码就会输出一个文本, 其后是一个数字, 再后面又有另一个文本。

```
std::cout<< "There are" << 219 << "ways I love you." << std::endl;
```

std::endl 表示一个行尾字符。输出流遇到 std::endl 时, 它会将到目前为止发送到槽中的所有内容统统输出, 并移至下一行。还有一种方法可表示行尾, 即使用 ‘\n’ 字符。\\n 字符是一个转义字符 (escape character), 它表示一个换行字符。转义字符可以用在任何加引号的文本串中。以下列出了一些最常用的转义字符。

- \n 换行
- \r 回车
- \t 跳格 (制表符)
- \\ 反斜线字符
- " 引号



流还可以用于接受用户的输入。为此，最简单的方法就是对输入流使用“>>”操作符。std::cin 输入流可以接受用户的键盘输入。用户输入可能很复杂，因为你并不知道用户会输入什么样的数据。有关如何使用输入流，请见第 14 章的详细解释。

### 1.1.2 命名空间

命名空间 (namespace) 要解决不同代码部分之间的命名冲突问题。例如，你可能在编写一段代码，其中包括一个名为 foo() 的函数。后来有一天，你又决定开始使用一个第三方库，而这个第三方库中也包括一个 foo() 函数。编译器看到代码中的 foo() 时，无从知道你指的是哪一个版本。一方面你无法修改第三方库中的函数名，另一方面，要修改你自己的函数名可能也是困难重重。

命名空间的出现使我们得以从这种尴尬境地中摆脱出来，因为这样就可以定义上下文，并在特定的上下文中定义名字。要把代码置于命名空间中，只需要将其包括在一个命名空间块中即可：

```
// namespaces.h
```

```
namespace mycode {  
    void foo();  
}
```

方法或函数的实现也可以在命名空间中处理：

```
// namespaces.cpp
```

```
#include <iostream>  
#include "namespaces.h"  
  
namespace mycode {  
    void foo() {  
        std::cout << "foo() called in the mycode namespace" << std::endl;  
    }  
}
```

通过把自己的 foo() 函数放到命名空间“mycode”中，就可把它与第三方库提供的 foo() 函数区分开来。要调用置于命名空间中的 foo() 版本，可以在函数名前面加上命名空间，如下所示。

```
mycode::foo(); // Calls the "foo" function in the "mycode" namespace
```

落在“mycode”命名空间块中的任何代码都可以直接调用同一个命名空间中的其他代码，而无需显式加上命名空间前缀。这种隐含的命名空间很有用，可以使代码更简洁、更具可读性。通过使用 using 指令，也可以不加命名空间前缀。这个指令告诉编译器其后的代码会利用指定命名空间中的名字。因此以下代码的命名空间就是隐含的：

```
// usingnamespaces.cpp
```

```
#include "namespaces.h"  
  
using namespace mycode;  
  
int main(int argc, char** argv)  
{  
    foo(); // Implies mycode::foo();  
}
```

一个源文件可以包含多个 using 指令，尽管这是一条捷径，但要注意不要滥用。在极端的情况下，如果通过 using 指令声明要使用已知的所有命名空间，实际上就会使命名空间根本无法起作用！如果用 using 指令使用了两个命名空间，而这两个命名空间中包含有相同的名字，那么还会导致命名冲突。需要知道你的代码是在哪个命名空间中操作，这一点也很重要，如此就能避免偶尔犯错，不至于调用了不正确的同名函数。

前面已经介绍了命名空间的语法，下面我们将在 Hello, World 程序中使用命名空间。cout 和 endl 实际上是定义于 std 命名空间中的名字。我们可以使用 using 指令重写 Hello, World 程序，如下所示：

```
// helloworld.cpp

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    cout << "Hello, World!" << endl;

    return 0;
}
```

using 指令还可以用于指示一个命名空间中的某个特定项。例如，如果只要使用 std 命名空间中的 cout，就可以如下指出：

```
using std::cout;
```

其后的代码可以直接引用 cout，而无需加命名空间前缀，不过如果要引用 std 命名空间中的其他项，还是需要显式地指出命名空间 (std)：

```
using std::cout;

cout << "Hello, World!" << std::endl;
```

### 1.1.3 变量

在 C++ 中，可以在代码中的任何位置声明变量 (variable)，而且一旦声明，当前块中在声明行之后的任何位置都可以使用该变量。实际上，工程小组应当确定是在每个函数的开始处声明变量，还是根据需要在不定的位置上声明。声明变量时，可以不为变量提供值。这种未赋值的变量往往会有一个“半随机”的值，这取决于当前内存中相应位置上恰好是什么值，而这往往会带来大量的 bug。C++ 中，变量在声明时可以为之赋一个初始值。以下代码显示了这两种形式的变量声明，两个变量声明都使用了表示整数值的 int。

```
// hellovariables.cpp

#include <iostream>

using namespace std;

int main(int argc, char** argv)
```

```

{
    int uninitializedInt;
    int initializedInt = 7;

    cout << uninitializedInt << " is a random value" << endl;
    cout << initializedInt << " was assigned an initial value" << endl;

    return (0);
}

```

运行这段代码时，第一行会输出一个随机值（取自内存），第二行则会输出数字 7。这段代码还展示了输出流如何使用变量。表 1-2 显示了 C++ 中最常用的一些变量类型。

表 1-2

类 型	描 述	用 法
int	正负整数（范围取决于编译器设置）	int i = 7;
short	短整数（通常为 2 字节）	short s = 13;
long	长整数（通常为 4 字节）	long l = -7;
unsigned int	将前 3 种类型（int、short 和 long）限制为值 $\geq 0$ （即非负）	unsigned int i = 2;
unsigned short		unsigned short s = 23;
unsigned long		unsigned long l = 5400;
float	单精度和双精度浮点数	float f = 7.2;
double		double d = 7.2
char	单字符	char ch = 'm';
bool	true 或 false（等价于非 0 或 0）	bool b = true;

C++ 没有提供基本的字符串类型。不过，在标准库中提供了字符串的一个标准实现，有关内容见本章后面及第 13 章的介绍。

通过对变量进行强制类型转换（casting），可以将其转换为其他类型。例如，一个 int 可以强制转换为一个 bool。C++ 提供了三种方法，可以显式地转换变量的类型。第一种方法沿袭自 C，不过这也是最常用的一种方法。第二种方法乍一看似乎更自然一些，但很少使用。第三种方法最详细，不过通常认为这种方法最清晰。

```
bool someBool = (bool)someInt;           // method 1
```

```
bool someBool = bool(someInt);           // method 2
```

```
bool someBool = static_cast<bool>(someInt); // method 3
```

如果这个整数为 0，其结果将是 false，否则为 true。在某些上下文中，变量会自动强制转换（coerced）。例如，short 可以自动转换为一个 long，因为 long 表示同一类型的数据，而且精度更高。

```
long someLong = someShort;                // no explicit cast needed
```

在自动强制转换变量时，需要注意有可能会丢失数据。例如，将一个 float 强制转换为 int 时，就会

丢掉一些信息（数字的小数部分）。如果将一个 float 赋给一个 int，而未做显式转换，许多编译器都会发出警告。如果你能肯定赋值表达式的左边类型与右边类型完全兼容，采用隐含的强制转换也是可以的。

#### 1.1.4 操作符

如果没有办法修改变量，变量能有什么用呢？表 1-3 列出了 C++ 中最常用的一些操作符（operator），并给出了一些示例代码来说明如何使用这些操作符。需要注意，C++ 中的操作符可以是二元操作符（binary，即操作两个变量），一元操作符（unary，即操作一个变量），甚至可以是三元操作符（ternary，操作三个变量）。C++ 中只有一个三元操作符，我们将在 1.1.6 节介绍。

表 1-3

操 作 符	描 述	用 法
=	这是一个二元操作符，可以把右边的值赋给左边的变量	<pre>int i; i = 3; int j; j = i;</pre>
!	这是一个一元操作符，可以对变量的真值取反（true/false 或非 0/0）	<pre>bool b = !true; bool b2 = !b;</pre>
+	这是一个完成加法的二元操作符	<pre>int i = 3 + 2; int j = i + 5; int k = i + j;</pre>
- * /	这些是完成减法、乘法和除法的二元操作符	<pre>int i = 5 - 1; int j = 5 * 2; int k = j / i;</pre>
%	这是一个取余数的二元操作符。也记作 mod（取模）操作符	<pre>int remainder = 5 % 2;</pre>
++	这是一个一元操作符，可使变量自增 1。如果这个操作符出现在变量的后面，表达式的结果则为自增之前的值（即先返回原值，再自增）。如果操作符出现在变量的前面，表达式的结果则是自增之后的新值（即先自增，再返回新值）	<pre>i++ ; ++ i;</pre>
--	这是一个一元操作符，可使变量自减 1	<pre>i-- ; -- i;</pre>
+=	i = i + j 的简写语法形式	i += j;
-=	这三个操作符分别是以下赋值表达式的简写语法形式： i = i - j;	i -= j;
*=	i = i * j;	i *= j;
/=	i = i / j;	i /= j;
%=	i = i % j;	i %= j;
&	取一个变量的原（二进制）位与另一个变量完成位“与”操作	i = j & k;

(续)

操 作 符	描 述	用 法
&=		j &= k;
	取一个变量的原（二进制）位与另一个变量完成位“或”操作	i = j   k; j  = k;
<<	取一个变量的原（二进制）位，将每一位左移（<<）或右移（>>）指定位数	i = i << 1;
>>		i = i >> 4;
<<=		i <<= 1;
>>=		i >>= 4;
^	对两个参数（操作数）完成位“异或”操作	i = i ^ j;
^=		i ^= j;

以下程序显示了最常用的一些变量类型和操作符的实际使用。如果不能确定这些变量和操作符会有什么表现，可以先自己考虑一下这个程序会有什么样的输出，然后再运行程序，验证答案是否正确。

```
// typetest.cpp

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    int someInteger = 256;
    short someShort;
    long someLong;
    float someFloat;
    double someDouble;

    someInteger++;
    someInteger *= 2;
    someShort = (short)someInteger;
    someLong = someShort * 10000;
    someFloat = someLong + 0.785;
    someDouble = (double)someFloat / 100000;

    cout << someDouble << endl;
}
```

C++ 编译器对于表达式计算的顺序有自己的一套规则。如果你写了一行复杂的代码，其中包括多个操作符，究竟以何种顺序执行可能并不明确。出于这个原因，更好的做法可能是把一条复杂的语句分解为多条较小的语句，或者使用小括号显式地将表达式分组处理。例如，除非你对 C++ 操作符优先表烂熟于胸，否则肯定对下面这行代码不太清楚：

```
int i = 34 + 8 * 2 + 21 / 7 % 2;
```

通过增加小括号，就可以清楚地看出哪些操作符先执行：

```
int i = 34 + (8 * 2) + ((21 / 7) % 2);
```

还可以把这条语句分解为多个单独的代码行，这样会更清楚一些：

```
int i = 8 * 2;  
int j = 21 / 7;  
j %= 2;  
i = 34 + i + j;
```

如果你自己试验这些方法，会发现这三种方法是等价的，都会得到同一个结果：i 等于 51。如果你认为 C++ 会从左到右计算表达式，就会得到错误的答案（认为 i 等于 1）。（译者注：不过，只要会加减乘除，应该不会犯这种低级错误吧。）实际上，C++ 会先计算 /、\* 和 %（按从左到右的顺序），再完成加法和减法操作，然后是位操作。通过加小括号可以显式地告诉编译器某个操作应当单独地计算。

### 1.1.5 类型

在 C++ 中，可以使用基本类型（int、bool 等）来构建自己设计的更复杂的类型。一旦你成为有一定经验的 C++ 程序员，一般就不太会使用以下技术了（这些都是从 C 照搬来的），因为可以使用类，而类的功能要强大得多。不过，还是要知道有这样两种构建类型的最常用的方法，这很重要，这样你就能看懂程序的语法了。

#### 枚举类型

整数实际上表示的是序列（数字序列）中的值。枚举类型（Enumerated type）则允许你定义自己的序列，这样就能用该序列中的值来声明变量。例如，在一个象棋（chess）程序中，可以把每个棋子表示为一个 int，对应不同的棋子类型有不同的常量值，如以下代码所示。表示类型的整数标记为 const，这说明这些整数不会改变。

```
const int kPieceTypeKing = 0;  
const int kPieceTypeQueen = 1;  
const int kPieceTypeRook = 2;  
const int kPieceTypePawn = 3;  
//etc.  
  
int myPiece = kPieceTypeKing;
```

这种表示不算不好，不过它有可能会变得很危险。因为棋子只是一个 int，如果另一个程序员增加了一些代码，要让棋子的值自增，那会有什么后果呢？如果加 1，原来的“国王”会变成“王后”，实际上这没有任何意义。更糟糕的是，有人可能会冒冒失失地给棋子赋一个值 -1，而并没有与 -1 相对应的常量。

枚举类型可以解决这些问题，它严格地为变量定义了一个值域。以下代码声明了一个新的类型 PieceT，它有 4 个可能的值，分别表示 4 种不同类型的棋子。

```
typedef enum { kPieceTypeKing, kPieceTypeQueen, kPieceTypeRook,  
              kPieceTypePawn  
} PieceT;
```

实际上，枚举类型也只是一个整数值而已。kPieceTypeKing 的实际值就是 0。不过，通过对类型为 PieceT 的变量定义可能的值，如果试图对 PieceT 变量完成算术运算，或者想把 PieceT 变量当作整数使用，编译器就会提示一个警告或错误。以下代码先声明了一个 PieceT 变量，然后想把它用作一个整数，在大多数编译器上就会得到一个警告。



```
PieceT myPiece;
```

```
myPiece = 0;
```

### 结构

利用结构 (struct)，可以把一个或多个已有的类型封装到一个新的类型中。结构的一个经典例子就是数据库记录。如果你要建立一个人事系统，记录员工的信息，可能需要保存每个员工的名、姓、中间名、员工编号和工资。以下头文件显示了一个包含上述所有信息的结构。

```
// employeestruct.h
```

```
typedef struct {  
    char    firstInitial;  
    char    middleInitial;  
    char    lastInitial;  
    int     employeeNumber;  
    int     salary;  
} EmployeeT;
```

类型声明为 EmployeeT 的变量内置有上述所有字段 (field)。结构中的各个字段可以使用 “.” 符号来访问。下面的例子将创建一个员工的一条记录，然后将其输出。

```
// structtest.cpp
```

```
#include <iostream>  
#include "employeestruct.h"
```

```
using namespace std;
```

```
int main(int argc, char** argv)
```

```
{  
    // Create and populate an employee.  
    EmployeeT anEmployee;
```

```
    anEmployee.firstInitial = 'M';  
    anEmployee.middleInitial = 'R';  
    anEmployee.lastInitial = 'G';  
    anEmployee.employeeNumber = 42;  
    anEmployee.salary = 80000;
```

```
    // Output the values of an employee.
```

```
    cout << "Employee: " << anEmployee.firstInitial <<  
        anEmployee.middleInitial <<  
        anEmployee.lastInitial << endl;  
    cout << "Number: " << anEmployee.employeeNumber << endl;  
    cout << "Salary: $" << anEmployee.salary << endl;
```

```
    return 0;  
}
```

### 1.1.6 条件语句

利用条件语句，可以根据某个条件是否为“真” (true) 来执行代码。C++ 中主要有 3 种条件语句。

### if/else 语句

最常见的条件语句就是 if 语句，后面可能跟有一个 else 语句（译者注，一般也称条件语句中包括 if 子句和 else 子句）。如果 if 语句内给定的条件为 true，就会执行 if 语句中相应的代码行或代码块。如果条件不为 true，而且存在 else 语句，就会执行 else 分支下的代码；倘若没有 else 语句，则执行该条件语句之后的代码。以下伪代码显示了一个层联嵌套的 if 语句，这是指 if 语句有一个 else 语句，而该 else 语句中又包括另一个 if 语句，如此继续。

```
if (i > 4) {  
    // Do something.  
} else if (i > 2) {  
    // Do something else.  
} else {  
    // Do something else.  
}
```

if 语句的小括号之间必须是一个 Boolean 值表达式，或者，这个表达式必须能计算为一个 Boolean 值。条件操作符（后面将介绍）提供了一些计算表达式的方法，从而可以得到一个 Boolean 值（true 或 false）。

### switch 语句

switch 语句也是一种基于一个变量的值来完成某些动作的条件语句。在 switch 语句中，变量必须与一个常量进行（相等）比较，上例中的 if 语句判断的条件是“大于”，因此不能转换为 switch 语句。每个常量值均表示一种“情况”，如果变量与某个情况（case）匹配，就会执行后面的代码行，一直到 break 语句为止。还可以提供一种默认（default）情况，如果所有其他情况都未能匹配，就会匹配这种默认情况（即执行相应的代码行）。

如果想基于一个变量的特定值做某件事情，而不是要对变量做某种测试，通常就可以使用 switch 语句。以下伪代码显示了 switch 语句的一种常用用法。

```
switch (menuItem) {  
    case kOpenMenuItem:  
        // Code to open a file  
        break;  
    case kSaveMenuItem:  
        // Code to save a file  
        break;  
    default:  
        // Code to give an error message  
        break;  
}
```

如果忽略了 break 语句，那么无论后面的情况是否匹配，都将执行后续的相应代码。在有些情况下，这可能很有用，不过更多的情况下这样会带来 bug。

### 三元操作符

C++ 中只有一个带有 3 个参数的操作符，称为三元操作符（ternary operator）。这个三元操作符用作一种简写的条件表达式，形如“if [something] then [perform action], otherwise [perform some other action]”，含义是“如果 [怎样]，那么 [完成动作]，否则 [完成另一个动作]”。这个三元操作符表示为一个 ? 和一个 :。如果变量 i 大于 2，以下代码将输出“yes”，否则输出“no”。

```
std::cout << ((i > 2) ? "yes" : "no");
```

三元操作符的优点是它几乎可以出现在任何地方。在前面的例子中，三元操作符就用在完成输出的代码中。要想记住这个三元操作符的语法，一种简便的方法是可以这样来处理问号：即把问号之前出现的语句当作一个问题。例如，“i 大于 2 吗？如果是，结果就是 ‘yes’；否则，结果就是 ‘no’”。

与 if 语句或 switch 语句不同，三元操作符并不根据结果来执行代码块。实际上，如前例所示，三元操作符要在代码中使用。这么说来，它就是一个操作符（如 + 和 -），而不是真正的条件语句（如 if 语句和 switch 语句）。

### 条件操作符

前面已经看到了一个条件操作符，但还没有为它提供正式的定义。“>”操作符用于比较两个值。如果左边的值大于右边的值，结果就是“true”。所有条件操作符都遵循这种模式，条件操作符的结果都是 true 或 false。

表 1-4 显示了另外一些常见的条件操作符。

表 1-4

操 作 符	描 述	用 法
< <= > >=	确定左边是否小于、小于或等于、大于、大于或等于右边	if (i <= 0) { std::cout << "i is negative"; }
==	确定左边是否等于右边。不要与=(赋值)操作符混淆!	if (i == 3) { std::cout << "i is 3"; }
!=	不等于。如果左边不等于右边，语句的结果为 true	if (i != 3) { std::cout << "i is not 3"; }
!	逻辑非。将布尔 (Boolean) 表达式的真值 (true/false) 取反。这是一个一元操作符	if (! someBoolean) { std::cout << "someBoolean is false"; }
&&	逻辑与。如果表达式的左右两边 (操作数) 都为 true，结果才为 true	if (someBoolean && someOtherBoolean) { std::cout << "both are true"; }
	逻辑或。如果表达式的某一边 (某一个操作数) 为 true，结果即为 true	if (someBoolean    someOtherBoolean) { std::cout << "at least one is true"; }

C++ 在计算表达式时使用一种“短路逻辑” (short-circuit logic)。这说明，一旦能够确定最后结果，表达式中的余下部分就不必再计算了。例如，在做以下多个布尔表达式的逻辑或操作时，只要发现其中一个布尔表达式为 true，就可以确定结果肯定为 true。那么余下的布尔表达式就根本不需要检查了。

```
bool result = bool1 || bool2 || (i > 7) || (27 / 13 % i + 1) < 2;
```

在上述例子中，如果发现 bool1 为 true，那么整个表达式必然为 true，因此其他部分就不计算了。通过这种方式，C++ 语言可以让你的代码少做一些不必要的工作。不过，如果后面的布尔表达式会以某种方式影响程序的状态（例如，可能调用一个单独的函数），这种短路逻辑就会带来 bug，而且这种 bug 很

难发现。以下代码显示了一条使用 && 操作符的语句，计算到第二项之后，根据短路逻辑就会停止计算，因为 0 总是计算为 false。

```
bool result = bool1 && 1 && (i > 7) && !done;
```

### 1.1.7 循环

计算机最擅长的就是一遍一遍地反复做同一件事。C++ 提供了三种循环结构。

#### while 循环

只要表达式计算为 true，while 循环 (while loop) 就允许反复地执行一个代码块。例如，以下这段“傻乎乎”的代码会把“This is silly.”输出 5 次。

```
int i = 0;
while (i < 5) {
    std::cout << "This is silly." << std::endl;
    i++;
}
```

可以在循环中使用关键字 break 直接从循环中跳出，并继续完成程序的执行。还可以使用关键字 continue 直接返回到循环的开始处，重新计算 while 表达式。使用这两个关键字都不能算是好的编程风格，因为这会导致程序的执行出现随意性的跳转。

#### do/while 循环

C++ 还有另外一种 while 循环，称为 do/while 循环。其工作原理与 while 循环很相似，只不过这里会先执行代码，在最后才做条件检查，确定是否继续循环。如果希望某个代码块起码要执行一次，而且根据某个条件还可能执行多次，就可以采用这种方式使用 do/while 循环。下面的例子至少会输出一次“This is silly.”，即使条件最后计算为 false 也不会妨碍执行。

```
int i = 100;
do {
    std::cout << "This is silly." << std::endl;
    i++;
} while (i < 5);
```

#### for 循环

for 循环 (for loop) 提供了另一种循环语法。所有 for 循环都可以转换为 while 循环，反之亦然。不过，for 循环的语法通常更方便一些，因为它给出了一个起始表达式、一个结束条件以及每次迭代最后所要执行的一条语句，并按这种方式进行循环。在下面的代码中，i 初始化为 0，只要 i 小于 5，循环就会持续下去，在每次迭代的最后，i 会自增 1。这段代码与上面的 while 循环可谓异曲同工，不过，对于某些程序员来说，这段代码读起来可能更容易一些，因为开始值、结束条件和每次迭代后执行的语句都放在同一行上。

```
for (int i = 0; i < 5; i++) {
    std::cout << "This is silly." << std::endl;
}
```

### 1.1.8 数组

数组 (array) 保存一系列的值, 所有值的类型都相同, 每个值可以通过它在数组中的位置加以访问。在 C++ 中, 声明数组时必须提供数组的大小。不能用变量给出大小, 数组大小必须是常量值。以下代码首先声明了一个大小为 10 个整数的数组, 其后利用 for 循环将数组中的每个整数初始化为 0。

```
int myArray[10];  
  
for (int i = 0; i < 10; i++) {  
    myArray[i] = 0;  
}
```

前面的例子显示了一个一维数组, 可以把它认为是一行整数, 每个整数都有自己的“车厢”, 这些车厢都是编了号的。C++ 还支持多维数组。可以把二维数组想像成一个棋盘, 棋盘上的每个位置都有一个坐标, 即在 x 轴上有一个 x 位置, 在 y 轴上有一个 y 位置。要画出三维或更高维的数组就比较困难了, 而且高维数组也很少使用。以下代码显示了如何分配二维数组, 这里为 Tic-Tac-Toe (连珠游戏) 棋盘分配了一个二维的字符数组, 然后在最中间的方格中放入一个“o”。

```
char ticTacToeBoard[3][3];  
  
ticTacToeBoard[1][1] = 'o';
```

图 1-1 显示了这个 Tic-Tac-Toe (连珠游戏) 棋盘, 在此可以看到每个方格的位置。

ticTacToeBoard[0][0]	ticTacToeBoard[0][1]	ticTacToeBoard[0][2]
ticTacToeBoard[1][0]	ticTacToeBoard[1][1]	ticTacToeBoard[1][2]
ticTacToeBoard[2][0]	ticTacToeBoard[2][1]	ticTacToeBoard[2][2]

图 1-1

C++ 中, 数组的第一个元素位置为 0 (即从 0 计起), 而不是 1! 数组的最后一个位置必然是数组的大小减 1。

### 1.1.9 函数

对于规模很大的程序, 如果把所有代码都放在 main() 中, 最后肯定是混乱不堪, 无法管理。为了使

程序易于理解，需要把代码分解（decompose）到简洁的函数中。

在C++中，首先要声明一个函数，使得其他代码可以使用这个函数。如果函数只是在一个特定的代码文件中使用，通常可以在源文件中同时声明和定义此函数。如果函数会由其他模块或文件使用，往往需要把函数声明放在头文件中，而定义放在源文件中。

函数声明通常称为“函数原型”（function prototype）或“签名”（signature），以此强调它们表示的是将如何访问函数，而不是如何实现函数（即实现函数的代码是怎样的）。

以下代码显示了一个函数。这个例子的返回类型为 void，这表示该函数不会为调用者提供任何结果。函数调用者必须为函数提供两个实参，即这个函数需要用到一个整数和一个字符。

```
void myFunction (int i, char c);
```

如果有上述函数声明，但是没有与之对应的具体函数定义，编译时到连接阶段就会报错，因为使用函数 myFunction() 的代码将调用根本不存在的代码。

以下函数定义只是打印出两个参数的值。

```
void myFunction(int i, char c)
{
    std::cout << "the value of i is " << i << std::endl;
    std::cout << "the value of c is " << c << std::endl;
}
```

在程序中的其他位置也可以调用 myFunction()，并为这两个参数传入常量或变量。下面显示了一些函数调用示例。

```
myFunction(8, 'a');
myFunction(someInt, 'b');
myFunction(5, someChar);
```

与C不同，在C++中，无参数的函数可以有一个空的参数列表。没有必要使用“void”来指示函数不带任何参数。不过，如果不返回任何值，仍需要使用“void”来指出。

C++函数还可以向调用者返回一个值。以下是一个函数的声明和定义，该函数完成两个数的相加，并返回结果。

```
int addNumbers(int number1, int number2);

int addNumbers(int number1, int number2)
{
    int result = number1 + number2;
    return (result);
}
```

### 1.1.10 结束语

到目前为止，你已经了解了C++编程的基本要领。如果本节还只是触及皮毛，请继续阅读1.2节。



来“领教”一些更高级、更深入的内容。如果你读本节时有困难，在看后面的内容之前，可能需要参考附录 B 中提到的一些很好的 C++ 入门书籍。

## 1.2 C++ 进阶

循环、变量和条件语句，这些都是很好的构造模块，不过我们需要了解的远非仅限于此。接下来将介绍 C++ 的一些特性，这些特性有助于 C++ 程序员编写（更好的）代码，除此以外，我们还将介绍另外一些特性，与这些特性提供的帮助相比，它们带来的混乱可能更为突出。如果你是一位 C 程序员，有多少 C++ 编程经验，就应当仔细地阅读本节。

### 1.2.1 指针和动态内存

基于动态内存，可以使用“动态”数据（编译时数据的大小并不固定）来构建程序。大多数相对复杂的程序都以某种方式使用了动态内存。

#### 栈和堆

C++ 应用中的内存分为两部分，一部分是栈（stack，也称堆栈），另一部分是堆（heap）。可以把栈看作是一摞卡片，最上面的卡片表示程序的当前作用域，这往往就是当前正在执行的函数。当前函数中声明的所有变量都置于栈顶帧中，即占用栈顶帧的内存，这就相当于一摞卡片中最上面的一张卡片。如果当前函数调用了另一个函数，举例来说，当前函数 `foo()` 调用了另一个函数 `bar()`，就会在这摞卡片上再加上一个新卡片，这样 `bar()` 就有了自己的栈帧（stack frame）以供使用。从 `foo()` 传递到 `bar()` 的所有参数都会从 `foo()` 栈帧复制到 `bar()` 栈帧中。参数传递的机制以及栈帧的有关内容将在第 13 章介绍。

图 1-2 显示了执行一个假想的 `foo()` 函数时，栈大致是什么样子，这个函数声明了两个整数值。

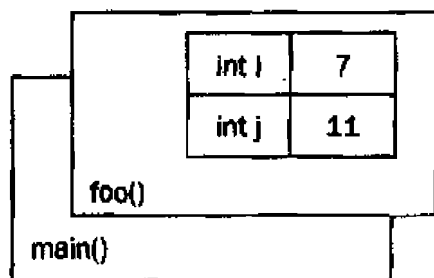


图 1-2

栈帧很有意义，因为栈帧可以为每个函数提供一个独立的内存工作区。如果一个变量是在 `foo()` 栈帧中声明的，那么调用 `bar()` 函数不会对它带来改变，除非你专门要求修改这个变量。另外，`foo()` 函数运行结束时，栈帧即消失，该函数中声明的所有变量都不会再占用内存了。

堆（heap）是一段完全独立于当前函数或栈帧的内存区。如果一个函数中声明了一些变量，而且希望当这个函数完成时其中声明的变量仍然存在，就可以将这些变量置于堆中。堆与栈相比，没有那么清晰的结构性。可以把堆看作是一“堆”小玩艺。程序可以在任何时刻向这个“堆”增加新的东西，或者修改“堆”中已经有的东西。

#### 动态分配的数组

根据栈的工作原理，编译器在编译时就必然能够确定每个栈帧有多大。由于栈帧大小是预定的，因此无法声明一个大小可变的数组。以下代码就无法通过编译，因为 `arraySize` 是变量，而不是常量。

```
int arraySize = 8;
int myVariableSizedArray[arraySize]; // This won't compile!
```

由于整个数组都要放在栈上，编译器需要准确地知道数组的大小，所以不允许用变量来指定数组大小。如果通过使用动态内存（dynamic memory），把数组放在堆中（而不是栈中），也可以在运行时才指定数组的大小。

有些 C++ 编译器确实支持超前声明（也称为前置声明），不过如今 C++ 规范中并不包括这个内容。大多数编译都提供了一种“严格”模式，它会“关掉”对 C++ 语言的这些非标准扩展。

要动态地分配一个数组，首先需要声明一个指针（pointer）：

```
int* myVariableSizedArray;
```

int 类型后面有一个 \*，它指出所声明的变量指向堆中某个存放整数的内存空间。可以把指针认为是一个箭头，指向动态分配的堆内存。到目前为止，这个变量并没有具体指向任何东西，因为你还没有给它赋任何值，它现在还是一个未初始化的变量。

要把指针初始化为新的堆内存，需要使用 new 命令：

```
myVariableSizedArray = new int [arraySize];
```

这会根据 arraySize 变量为足够多的整数分配内存。图 1-3 显示了执行这段代码之后栈和堆是什么样子。可以看到，指针变量仍然位于栈中，但是数组则是在堆上动态创建的。

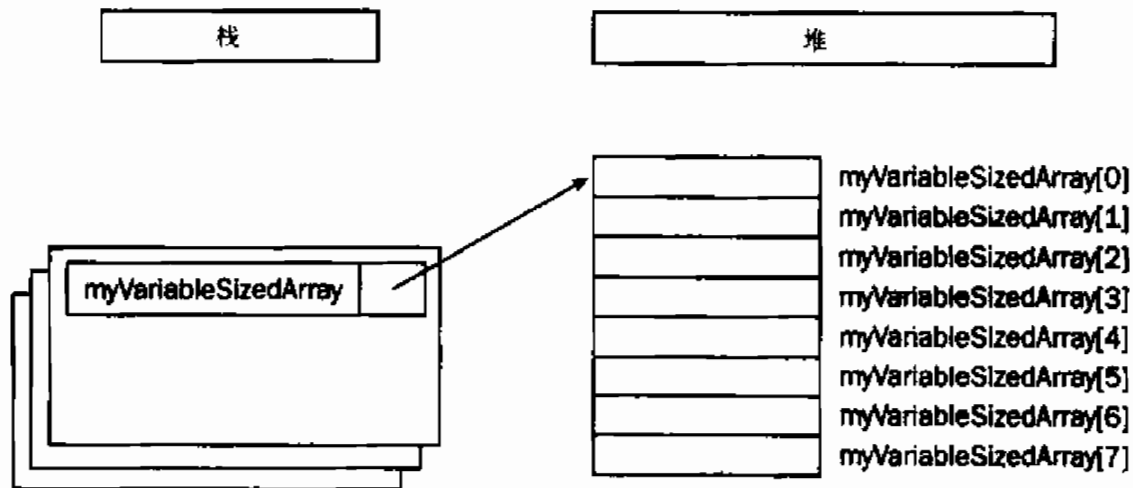


图 1-3

既然已经分配了内存，就可以把 myVariableSizedArray 当成一个基于栈的常规数组来使用了：

```
myVariableSizedArray [3] = 2;
```

代码使用完这个数组之后，应当将数组从堆中删除，以便其他变量利用该数组原先占用的内存。为此，在 C++ 中，要使用 delete 命令。

```
delete [] myVariableSizedArray;
```

delete 后面的中括号指示出正在删除一个数组。

C++ 的 `new` 和 `delete` 命令与 C 中的 `malloc()` 和 `free()` 很相似。不过, `new` 和 `delete` 的语法更简单一些, 因为你不必知道需要多少字节的内存空间。

### 使用指针

除了动态分配数组, 使用堆内存还有另外的一些原因。通过采用一种与动态分配数组类似的语法, 还可以将任何变量置于堆中:

```
int* myIntegerPointer = new int;
```

在这种情况下, 该指针只是指向一个单独的整数值。要访问这个值, 需要对指针解除引用 (dereference)。可以这样来理解解除引用, 即沿着指针的箭头找到堆中的实际值。如果要对堆中这个新分配的整数设置值, 可以使用如下的代码:

```
*myIntegerPointer = 8;
```

需要注意, 这与将 `myIntegerPointer` 的值设置为 8 有所不同。在此没有改变指针, 所改变的只是该指针指向的内存空间。如果是对指针重新赋值 (即 `myIntegerPointer = 8;`), 这个指针就会指向内存地址 8, 而这里可能只是一个随机的垃圾, 最终有可能导致程序终止。

指针并不总是指向堆内存空间。也可以声明一个指针指向栈中的某个变量, 甚至可以指向另一个指针。要得到指向某变量的指针, 可以使用 `&` (取地址) 操作符:

```
int i = 8;  
int* myIntegerPointer = &i; // Points to the variable with the value 8
```

对于处理指向结构的指针, C++ 有一种特殊的语法。理论上讲, 如果有一个指向某结构的指针, 可以先用 `*` 将该指针解除引用, 然后使用正常的 “.” 语法来访问结构中的字段, 如以下代码所示, 在此假设存在一个名为 `getEmployee()` 的函数。

```
EmployeeT* anEmployee = getEmployee();  
cout << (*anEmployee).salary << endl;
```

这种语法有点乱。利用 `->` (箭头) 操作符, 可以一步做两个工作, 既对指针解除引用, 又完成字段访问。以下代码与前面的代码等价, 但阅读起来更为清晰。

```
EmployeeT* anEmployee = getEmployee();
```

```
cout << anEmployee->salary << endl;
```

正常情况下, 向函数传递一个变量时, 都是在按值传递 (passing by value)。如果函数取一个整数参数, 实际上取的是传入整数的一个副本。在 C 中经常会使用指向栈变量的指针, 从而使函数可以修改其他栈帧中的变量, 这实际上就是按引用传递 (passing by reference)。通过对指针解除引用, 即使变量不在当前的栈帧中, 函数也可以对表示变量的内存加以修改。这在 C++ 中不太常见, 因为 C++ 有一种更好称为引用 (reference) 的机制, 我们将在后面介绍引用的有关内容。

### 1.2.2 C++ 中的字符串

C++ 中有三种使用文本串的方法, 分别是 C 风格、C++ 风格和一些非标准的方法。C 风格就是将字符串表示为字符数组; C++ 风格是把字符串表示包装在一种更易于使用的串类型中。

## C风格的字符串

形如“Hello, World”的文本串在内部表示为一个字符数组，并以字符‘\0’表示串结束。可以看到，数组和指针有时是相关的。可以使用其中任何一个（数组或指针）来表示一个字符串，如下所示：

```
char arrayString[20] = "Hello, World";  
char* pointerString = "Hello, World";
```

对于 arrayString，编译器会为它在栈中分配 20 个字符的空间。数组中前 13 个字符分别填以‘H’、‘e’等等字符，最后以字符‘\0’结束。位置 13 到 19 中的字符（译者注：对应数组中第 14 到第 20 个元素）包含的是一些随机值，即当前恰好位于相应内存空间中的值。根据‘\0’字符，使用字符串的代码就可以知道字符串的内容在哪里结束。即使数组长度为 20，处理或输出该字符串的函数都应当忽略‘\0’字符之后的所有内容。

对于 pointerString，编译器会在栈中分配足够的内存来存放指针（只是要存放一个指针）。这个指针指向编译器预留的一段内存区（用以保存常量串“Hello, World”）。在这个字符串中，‘d’字符后面也有一个‘\0’字符。

C 语言提供了许多处理字符串的标准函数，见<cstring>头文件。在此不会介绍这个标准库的细节，因为 C++ 提供了一种更简洁的方式来处理字符串。

## C++字符串

理解 C 风格的字符串很重要，因为 C++ 程序员还会经常使用这种语法。不过，C++ 提供了一种灵活得多的 string 类型。这种 string 类型（见<string>头文件）相当于一种基本类型。与 I/O 流类似，这种 string 类型也放在“std”包中。以下例子显示了可以像使用字符数组一样使用字符串。

```
// stringtest.cpp  
  
#include <string>  
#include <iostream>  
  
using namespace std;  
  
int main(int argc, char** argv)  
{  
    string myString = "Hello, World";  
  
    cout << "The value of myString is " << myString << endl;  
  
    return 0;  
}
```

C++ 字符串的神奇之处在于，你可以使用标准操作符来处理这些字符串。C 中往往需要使用某个函数来完成操作，如需要使用 strcat() 来连接两个串，在 C++ 中则没有这个必要，可以简单地使用+操作符完成串连接。如果想使用==操作符来比较两个 C 风格的字符串，就会发现不能如你所愿，如果对 C 风格的字符串使用==，所比较的并非字符串的内容，而是字符数组的地址。对于 C++ 字符串，== 则确实会比较两个串（的内容）。下面的例子显示了可以用于 C++ 字符串的一些标准操作符。

```
// stringtest2.cpp  
  
#include <string>  
#include <iostream>
```

```
using namespace std;

int main(int argc, char** argv)
{
    string str1 = "Hello";
    string str2 = "World";
    string str3 = str1 + " " + str2;

    cout << "str1 is " << str1 << endl;
    cout << "str2 is " << str2 << endl;
    cout << "str3 is " << str3 << endl;

    if (str3 == "Hello World") {
        cout << "str3 is what it should be." << endl;
    } else {
        cout << "Hmmm . . . str3 isn't what it should be." << endl;
    }

    return (0);
}
```

前面几个例子显示了 C++ 字符串的一些特性。第 13 章还将提供更详细的介绍。

### 非标准字符串

许多 C++ 程序员并不使用 C++ 风格的字符串，对此有多方面的原因。有些程序员只是不知道这种 string 类型的存在，因为它不一定是 C++ 规范的一部分。另外一些程序员经过多年的实践发现，string 类型的表现不尽如人意，因此他们开发了自己的串类型。也许最常见的原因是开发框架和操作系统在字符串表示方面都有自己的方式，如 Microsoft 的 MFC 就提供了 CString 类。一般地，这是为了做到向后兼容或者为了解决遗留问题。采用 C++ 启动一个项目时，一定要提前确定开发小组如何表示字符串，这一点很重要。

### 1.2.3 引用

大多数函数都有一个共同的模式，它们会取 0 个或多个参数，完成某些计算，并返回一个结果。不过，有时也可能打破这种模式。你可能想要返回两个值，或者希望函数能够改变传入的某个参数的值。

在 C 中，要达到这样一些目的，主要的做法就是传入变量的指针，而不是传入变量自身。这种方法惟一的缺陷在于，对于原本很简单的任务，可能会因此引入繁杂的指针语法。在 C++ 中，对“按引用传递”（即传引用）有一个明确的机制。如果对一个类型附加了 &，这就表示该变量是一个引用。仍然会把它当常规的变量来使用，不过就其本质而言，它实际上是原变量的一个指针。以下是一个 addOne() 函数的两种实现。第一个函数实现中，对传入的变量没有任何影响，因为这个参数是按值传递的。第二个实现使用了一个引用，因此会修改原来的变量。

```
void addOne(int i)
{
    i++; // Has no real effect because this is a copy of the original
}
```

```
void addOne(int& i)
{
    i++; // Actually changes the original variable
}
```



如果要基于一个整数引用来调用 `addOne()` 函数（即取一个引用作为参数的 `addOne()` 函数），与只取一个整数作为参数的 `addOne()` 函数相比，调用语法并无不同。

```
int myInt = 7;
addOne(myInt);
```

#### 1.2.4 异常

C++ 是一种非常灵活的语言，不过不是特别安全。你可以编写一些代码直接操纵随机的内存地址，甚至可以尝试除 0（计算机不能很好地处理无限性），编译器对于这些做法是默许的。为了增加 C++ 的安全性，引入的语言特性之一就是异常（exception）。

所谓异常就是一种意料之外的情况。例如，如果你编写了一个函数，要访问一个 Web 页面，那么在许多方面都可能出错。该页面所在的 Internet 主机可能关机，页面可能返回为空白页面，另外连接有可能断开。在许多程序设计语言中，对于这种情况的处理是从函数返回一个特殊的值，如 NULL 指针。异常则为处理这些问题提供了另一种机制，相比之下这种异常机制要好得多。

异常带来了一些新的术语。如果一段代码检测到一种异常的情况，就会抛出（throw）一个异常。另一段代码会捕获（catch）此异常，并且采取适当的动作。以下例子显示了一个 `divideNumbers()` 函数，如果调用者传入的分母为 0，这个函数就会抛出一个异常。

```
#include <stdexcept>

double divideNumbers(double inNumerator, double inDenominator)
{
    if (inDenominator == 0) {
        throw std::exception();
    }

    return (inNumerator / inDenominator);
}
```

执行到 `throw` 代码行时，函数会立即结束，不会返回值。如果调用者用一个 `try-catch` 块把这个函数调用包起来（如以下代码所示），调用者就会接收到这个异常，并且能够加以处理。

```
#include <iostream>
#include <stdexcept>

int main(int argc, char** argv)
{
    try {
        std::cout << divideNumbers(2.5, 0.5) << std::endl;
        std::cout << divideNumbers(2.3, 0) << std::endl;
    } catch (std::exception exception) {
        std::cout << "An exception was caught!" << std::endl;
    }
}
```

第一个 `divideNumbers()` 调用会成功地执行，结果将输出给用户。第二个调用则会抛出一个异常。此时不会返回任何值，惟一的输出就是捕获到异常时所打印的错误消息。上述代码块的输出如下：

```
5
An exception was caught!
```



C++ 中异常可能会变得很复杂。要想正确地使用异常，就需要了解抛出异常时栈变量会发生什么情况，而且必须相当仔细，需要正确地捕获和处理必要的异常。前面的例子使用了内置的 `std::exception` 异常类型，不过最好是编写自己的异常类型，对于所抛出的错误，你自己的异常可以更为特定、更加具体。不同于 Java 语言，C++ 编译器不要求捕获可能出现的每一个异常。（译者注：这句话不完全正确，因为在 Java 语言中，同样不要求捕获每一个异常，比如 Java 虚拟机抛出的 `Error`、诸如数组下标越界等系统运行时异常等等都无需 Java 程序捕获和处理）。

如果代码没有捕获任何异常，但是确实抛出了一个异常，这个异常就会被程序自身捕获，这会导致程序终止。有关异常的更复杂的情况将在第 15 章详细讨论。

### 1.2.5 const 的多重用途

C++ 中的关键字 `const` 可以有多种使用方式。尽管它的所有用途都是相关的，不过还是存在着细微差别。本书作者之一就发现，`const` 的这些细节很值得研究！在第 12 章中，你将了解 `const` 的所有用法。本小节只是大致介绍了它的一些最常见的用法。

#### const 常量

关键字 `const` 应该与常量有某种关联，这正是 `const` 的一个用途。在 C 语言中，程序员通常会使用预编译器 `#define` 机制（即预编译指令 `#define`）来声明符号名（即符号常量，其值在程序执行过程中不会发生改变），如版本号。在 C++ 中，则建议程序员尽量避免使用 `#define`，而倾向于使用 `const` 来定义常量。利用 `const` 定义常量与定义变量很类似，只不过编译器会确保代码不能修改此常量的值。

```
const float kVersionNumber = 2.0 ;
const string kProductName = "Super Hyper Net Modulator";
```

#### 使用 const 来保护变量

在 C++ 中，可以将一个未加 `const` 的变量强制转换为一个 `const` 变量。为什么要这样做呢？这样可以提供一定程度的保护，防止其他代码修改这个变量。如果调用一个函数，这个函数是你的一个同事编写的，而且你希望确保该函数不会改变传入的某个参数的值，就可以告诉你的同事，要求该函数取一个 `const` 参数。如果这个函数试图修改该参数的值，将不能通过编译。

在以下代码中，调用 `mysteryFunction()` 时，`char*` 参数会自动地强制转换为一个 `const char*`。如果编写 `mysteryFunction()` 的人想修改此字符数组中的值，这段代码就会编译失败。确实可以采取一些方法绕过这个限制，不过真的想“绕路而行”还是需要费点功夫的。C++ 只能做有限的保护，即只能防止无意地修改 `const` 变量。

```
// consttest.cpp
void mysteryFunction(const char* myString);

int main(int argc, char** argv)
{
    char* myString = new char[2];
    myString[0] = 'a';
    myString[1] = '\0';

    mysteryFunction(myString);

    return (0);
}
```

## const 引用

你可能经常会看到代码中使用了 const 引用参数。表面看来，这好像有些矛盾。引用参数允许你修改来自另一个上下文的变量的值，而 const 好像正是要避免这种修改。

const 引用参数的主要意义在于它能提高效率。向一个函数传递一个变量时，会建立一个完整的副本。如果传递一个引用，实际上只是传递了原变量的一个指针，这样计算机就无需为之建立副本。通过传递一个 const 引用，能尽享这两个方面的优势，一来不用建立副本，二来原变量也不会被修改。

如果要处理对象，const 引用就会更重要，因为对象可能相当庞大，而且建立对象的副本可能会导致预想不到的副作用。这些细节问题将在第12章介绍。

## 1.3 作为一种面向对象语言的 C++

如果你是一名 C 程序员，可能会认为本章迄今为止介绍的特性只是对 C 语言的一些便利扩充。顾名思义，在许多方面，C++ 这个语言只是一种“更好的 C”。但这个观点忽视了一个重要的方面。C++ 是不同于 C 的一种面向对象语言。

面向对象程序设计 (Object-oriented programming, OOP) 是一种迥然不同的编写代码的方法，而且可以证明这种方法更为自然。如果你习惯于使用诸如 C 或 Pascal 等过程性语言，也不必担心。第3章将介绍所有必要的背景知识，基于这些介绍，就能很好地改变观念，转向面向对象领域。如果你已经了解 OOP 理论，本节后面的内容将使你更多地了解（或者回想起）基本的 C++ 对象语法。

## 声明类

类 (class) 定义了一个对象的特征。这与 struct 有些类似，不过类除了定义属性外，还定义了行为。在 C++ 中，类通常在头文件中声明，而在相应的源文件中得到完全定义。

以下显示了一个机票类的基本类定义。这个类可以基于飞行里程以及顾客是否为“Elite Super Rewards Program”的会员来计算机票的价格。在此定义中，首先是声明类名，并在一对大括号中声明该类的数据成员（属性）及其方法（行为）。每个数据成员（data member）都与一个特定的访问层次（译者注：这也称为可见性修饰符）相关联：public、protected 或 private。这些修饰符可以以任何顺序出现，而且可以重复出现。

```
// AirlineTicket.h

#include <string>

class AirlineTicket
{
public:
    AirlineTicket();
    ~AirlineTicket();

    int    calculatePriceInDollars();

    std::string  getPassengerName();
    void        setPassengerName(std::string inName);
    int         getNumberOfMiles();
    void        setNumberOfMiles(int inMiles);
    bool        getHasEliteSuperRewardsStatus();
    void        setHasEliteSuperRewardsStatus(bool inStatus);
```

```

private:
    std::string mPassengerName;
    int         mNumberOfMiles;
    bool        fHasEliteSuperRewardsStatus;
};

```

与类同名而且没有返回类型的方法称为构造函数 (constructor)。创建这个类的对象时会自动调用构造函数。带~字符而且后面跟有类名的方法为析构函数 (destructor)。撤销对象时会自动调用析构函数。

下面的示例程序使用了上例中声明的类。这个例子显示了如何创建一个基于栈的 AirlineTicket 对象, 另外还显示了如何创建一个基于堆的 AirlineTicket 对象。

```

// AirlineTicketTest.cpp

#include <iostream>
#include "AirlineTicket.h"

using namespace std;

int main(int argc, char** argv)
{
    AirlineTicket myTicket; // Stack-based AirlineTicket

    myTicket.setPassengerName("Sherman T. Socketwrench");
    myTicket.setNumberOfMiles(700);
    int cost = myTicket.calculatePriceInDollars();
    cout << "This ticket will cost $" << cost << endl;

    AirlineTicket* myTicket2; // Heap-based AirlineTicket

    myTicket2 = new AirlineTicket(); // Allocate a new object
    myTicket2->setPassengerName("Laudimore M. Hallidue");
    myTicket2->setNumberOfMiles(2000);
    myTicket2->setHasEliteSuperRewardsStatus(true);
    int cost2 = myTicket2->calculatePriceInDollars();
    cout << "This other ticket will cost $" << cost2 << endl;
    delete myTicket2;

    return 0;
}

```

AirlineTicket 类方法的定义如下所示。

```

// AirlineTicket.cpp

#include <iostream>
#include "AirlineTicket.h"

using namespace std;

AirlineTicket::AirlineTicket()
{
    // Initialize data members
    fHasEliteSuperRewardsStatus = false;
    mPassengerName = "Unknown Passenger";
}

```

```
mNumberOfMiles = 0;
}

AirlineTicket::~AirlineTicket()
{
    // Nothing much to do in terms of cleanup
}

int AirlineTicket::calculatePriceInDollars()
{
    if (getHasEliteSuperRewardsStatus()) {
        // Elite Super Rewards customers fly for free!
        return 0;
    }

    // The cost of the ticket is the number of miles times
    // 0.1. Real airlines probably have a more complicated formula!
    return static_cast<int>((getNumberOfMiles() * 0.1));
}

string AirlineTicket::getPassengerName()
{
    return mPassengerName;
}

void AirlineTicket::setPassengerName(string inName)
{
    mPassengerName = inName;
}

int AirlineTicket::getNumberOfMiles()
{
    return mNumberOfMiles;
}

void AirlineTicket::setNumberOfMiles(int inMiles)
{
    mNumberOfMiles = inMiles;
}

bool AirlineTicket::getHasEliteSuperRewardsStatus()
{
    return (fHasEliteSuperRewardsStatus);
}

void AirlineTicket::setHasEliteSuperRewardsStatus(bool inStatus)
{
    fHasEliteSuperRewardsStatus = inStatus;
}
```

上述例子展示了创建和使用类的一般语法。当然，我们要学的还有很多。第8章和第9章将更为深入地讨论C++中有关定义类的特定机制。

## 1.4 你的第一个实用的C++程序

前面在讨论结构(struct)时举过一个员工数据库的例子，下面的程序将以此为基础。不过，现在你



会得到一个功能完备的 C++ 程序，它利用了本章讨论的许多特性。这个实际例子包括有类、异常、流、数组、命名空间、引用的使用，还使用了 C++ 的其他一些特性。

### 1.4.1 一个员工记录系统

要管理一个公司的员工记录，这个程序必须很灵活，而且需要提供一些有用的特性。程序应当包括以下功能：

- 能够增加一名员工
- 能够解雇一名员工
- 能够给员工加薪
- 能够查看所有员工，包括过往员工和目前的在位员工
- 能够查看所有在位员工
- 能够查看所有过往员工

设计程序时，将代码分为三个部分。Employee 类封装了描述一名员工的相关信息。Database 类要管理公司中的所有员工。另外有一个单独的 UserInterface 文件，这个文件提供了程序的交互功能。

### 1.4.2 Employee 类

Employee 类要维护有关一名员工的所有信息。利用这个方法，可以查询和修改员工的信息。Employee 还知道如何在控制台上显示自己的信息。另外还有一些方法可以调整员工的工资和雇用状态（在位还是解雇）。

Employee.h

Employee.h 文件声明了 Employee 类的行为。下面将分别介绍这个文件中的各个部分。

```
// Employee.h  
  
#include <iostream>  
  
namespace Records {
```

文件的前几行包括有一行注释，指示了这个文件的文件名，并指出要包含流功能。

这段代码还声明了包含在大括号中的后续代码存在于 Records 命名空间中。在这个程序中，特定于应用的代码使用的就是 Records 命名空间。

```
const int kDefaultStartingSalary = 30000;
```

这个常量表示新员工的默认最低工资，位于 Records 命名空间中。Records 命名空间中的其他代码可以简单地用 kDefaultStartingSalary 来访问这个常量。否则，其他命名空间中的代码必须将其引用为 Records::kDefaultStartingSalary。

```
class Employee  
{  
public:  
  
    Employee();  
  
    void    promote(int inRaiseAmount = 1000);  
    void    demote(int inDemeritAmount = 1000);
```

```

void    hire();    // Hires or rehires the employee
void    fire();    // Dismisses the employee
void    display(); // Outputs employee info to the console

// Accessors and setters
void    setFirstName(std::string inFirstName);
std::string getFirstName();
void    setLastName(std::string inLastName);
std::string getLastName();
void    setEmployeeNumber(int inEmployeeNumber);
int     getEmployeeNumber();
void    setSalary(int inNewSalary);
int     getSalary();
bool    getIsHired();

```

以上声明了 Employee 类，并且声明了它的公共方法。promote() 和 demote() 方法都取整数参数，而且为此参数指定了一个默认值。采用这种方式，其他代码就可以忽略该整数参数，在没有提供此参数的情况下，会自动地使用其默认值。

这里有许多存取方法（accessor），由此提供了必要的存取机制，可以修改员工的有关信息，或者查询一个员工的当前信息。

```

private:
    std::string mFirstName;
    std::string mLastName;
    int         mEmployeeNumber;
    int         mSalary;
    bool        fHired;
};

```

最后，数据成员声明为 private（私有），这样其他部分的代码就不能直接修改这些属性。只能通过存取方法这条公开的途径来修改或查询这些私有数据成员的值。

Employee.cpp

Employee 类方法的实现如下所示。

```

// Employee.cpp
#include <iostream>
#include "Employee.h"
using namespace std;
namespace Records {
    Employee::Employee()
    {
        mFirstName = "";
        mLastName = "";
        mEmployeeNumber = -1;
        mSalary = kDefaultStartingSalary;
        fHired = false;
    }
}

```

Employee 构造函数设置了 Employee 数据成员的初始值。默认地，新员工没有名字，员工号为-1，工资为默认的最低工资，状态为未雇用。

```
void Employee::promote(int inRaiseAmount)
{
    setSalary(getSalary() + inRaiseAmount);
}

void Employee::demote(int inDemeritAmount)
{
    setSalary(getSalary() - inDemeritAmount);
}
```

promote()和 demote()方法只是利用一个新值来调用 setSalary()方法。注意，这两个方法的整数参数有默认值，但是在源文件中并没有出现其默认值（只是头文件中有默认值）。

```
void Employee::hire()
{
    fHired = true;
}

void Employee::fire()
{
    fHired = false;
}
```

hire()和 fire()方法只是适当地设置 fHired 数据成员。

```
void Employee::display()
{
    cout << "Employee: " << getLastName() << ", " << getFirstName() << endl;
    cout << "-----" << endl;
    cout << (fHired ? "Current Employee" : "Former Employee") << endl;
    cout << "Employee Number: " << getEmployeeNumber() << endl;
    cout << "Salary: $" << getSalary() << endl;
    cout << endl;
}
```

display()方法使用控制台输出流来显示当前员工的有关信息。由于这个代码是 Employee 类的一部分，它可以直接访问数据成员（如 mSalary），而不必使用存取方法（如 getSalary()）。不过即使是同一个类中的代码，如果有存取方法，最好还是使用存取方法，一般认为这是一种好的编程风格。

```
// Accessors and setters

void Employee::setFirstName(string inFirstName)
{
    mFirstName = inFirstName;
}

string Employee::getFirstName()
{
    return mFirstName;
}

void Employee::setLastName(string inLastName)
```



```
{
    mLastName = inLastName;
}

string Employee::getLastName()
{
    return mLastName;
}

void Employee::setEmployeeNumber(int inEmployeeNumber)
{
    mEmployeeNumber = inEmployeeNumber;
}

int Employee::getEmployeeNumber()
{
    return mEmployeeNumber;
}

void Employee::setSalary(int inSalary)
{
    mSalary = inSalary;
}

int Employee::getSalary()
{
    return mSalary;
}

bool Employee::getIsHired()
{
    return fHired;
}
}
```

这里通过大量存取方法和设置方法来完成获取和设置值的简单任务。尽管这些方法看上去很简单，但这是很有意义的，也可以将数据成员置为公共，不过与这种做法相比，建立这样一些简单的存取方法和设置方法更有好处。以后还可以扩充这些存取方法和设置方法，例如，将来也许你会希望在 `setSalary()` 方法中完成上下界检查。

EmployeeTest.cpp

在写单个的类时，对其独立地进行测试往往很有好处。以下代码包括有一个 `main()` 函数，它会使用 `Employee` 类完成一些简单的操作。一旦确信 `Employee` 类能正常工作，就应当删除这个文件，或者将以下代码注释掉，以免编译代码时存在多个 `main()` 函数。

```
// EmployeeTest.cpp

#include <iostream>

#include "Employee.h"

using namespace std;
using namespace Records;
```

```

int main (int argc, char** argv)
{
    cout << "Testing the Employee class." << endl;
    Employee emp;

    emp.setFirstName("Marni");
    emp.setLastName("Kleper");
    emp.setEmployeeNumber(71);
    emp.setSalary(50000);
    emp.promote();
    emp.promote(50);
    emp.hire();
    emp.display();

    return 0;
}

```

### 1.4.3 Database 类

Database 类使用一个数组来保存 Employee 对象。这里使用一个名为 mNextSlot 的整数作为标记，用以跟踪下一个未用的数组“槽”。用这种方法来保存对象可能不算理想，因为数组的大小是固定的。在第 4 章和第 21 章中，你将了解到 C++ 标准库中提供的一些数据结构，使用这些结构保存对象更为合适。

Database.h

```

// Database.h

#include <iostream>
#include "Employee.h"

namespace Records {

    const int kMaxEmployees = 100;
    const int kFirstEmployeeNumber = 1000;
}

```

与数据库关联有两个常量。最大员工数是一个常量，这是因为记录保存在一个固定大小的数组中。由于数据库还需要负责自动地为新员工分配一个员工号，因此这里还定义了另一个常量，用以指出员工号从哪里开始计数。

```

class Database
{
public:
    Database();
    ~Database();

    Employee& addEmployee(std::string inFirstName, std::string inLastName);
    Employee& getEmployee(int inEmployeeNumber);
    Employee& getEmployee(std::string inFirstName, std::string inLastName);
}

```

这个数据库提供了一种增加新员工的简便方法，即通过提供员工的名和姓就可以增加新员工。为方便起见，这个方法将返回新员工的一个引用。外部代码还可以通过调用 getEmployee() 方法来得到一个员工引用。在此分别声明了这个方法的两个版本。一个方法允许通过员工号来获取员工引用，另一个方法

则需要一个名和一个姓作为参数。

```
void    displayAll();  
void    displayCurrent();  
void    displayFormer();
```

由于数据库是所有员工记录的集中存储库，它提供了一些方法，可以输出所有员工、目前在位的员工以及不再雇用的员工。

```
protected:  
    Employee    mEmployees[kMaxEmployees];  
    int         mNextSlot;  
    int         mNextEmployeeNumber;  
};  
}
```

mEmployees 数组是一个固定大小的数组，其中包含了 Employee 对象。创建数据库时，这个数组将填上没有名字的员工，而且所有员工的员工号都为 -1。调用 addEmployee() 方法时，会用实际数据填充这些空员工（译者注：这里的空员工是指员工对象无实际内容，而不是说员工对象为 null）。mNextSlot 数组成员用来跟踪下一次要填充哪一个空员工。mNextEmployeeNumber 数组成员则记录要为新员工分配的员号。

Database.cpp

```
// Database.cpp  
  
#include <iostream>  
#include <stdexcept>  
  
#include "Database.h"  
  
using namespace std;  
  
namespace Records {  
  
    Database::Database()  
    {  
        mNextSlot = 0;  
        mNextEmployeeNumber = kFirstEmployeeNumber;  
    }  
  
    Database::~Database()  
    {  
    }  
}
```

Database 构造函数负责将下一个槽（即下一次要填充的空员工，mNextSlot）和下一个员工号（mNextEmployeeNumber）初始化为起始值。mNextSlot 初始化为 0，这样增加第一个员工时，会将其放在 mEmployees 数组中下标为 0 的槽中。

```
Employee& Database::addEmployee(string inFirstName, string inLastName)  
{  
    if (mNextSlot >= kMaxEmployees) {  
        cerr << "There is no more room to add the new employee!" << endl;  
        throw exception();  
    }  
}
```

```

    }

    Employee& theEmployee = mEmployees[mNextSlot++];
    theEmployee.setFirstName(inFirstName);
    theEmployee.setLastName(inLastName);
    theEmployee.setEmployeeNumber(mNextEmployeeNumber++);
    theEmployee.hire();

    return theEmployee;
}

```

addEmployee()方法会用实际的信息填充下一个“空”员工。首先会做一个检查，以确保 mEmployees 数组未满，如果数组确实已经满了，则抛出一个异常。需要注意，使用完 mNextSlot 和 mNextEmployeeNumber 数据成员之后，这两个数据成员会自增，以使下一个员工得到一个新的槽和新的员工号。

```

Employee& Database::getEmployee(int inEmployeeNumber)
{
    for (int i = 0; i < mNextSlot; i++) {
        if (mEmployees[i].getEmployeeNumber() == inEmployeeNumber) {
            return mEmployees[i];
        }
    }

    cerr << "No employee with employee number " << inEmployeeNumber << endl;
    throw exception();
}

Employee& Database::getEmployee(string inFirstName, string inLastName)
{
    for (int i = 0; i < mNextSlot; i++) {
        if (mEmployees[i].getFirstName() == inFirstName &&
            mEmployees[i].getLastName() == inLastName) {
            return mEmployees[i];
        }
    }

    cerr << "No match with name " << inFirstName << " " << inLastName << endl;
    throw exception();
}

```

不论是哪一个版本的 getEmployee()，其工作方式都是类似的。这个方法会循环处理 mEmployees 数组中所有非空的员工，查看各 Employee 是否与向方法传入的信息相匹配。如果没有找到匹配的员工记录，会输出一个错误，并抛出一个异常。

```

void Database::displayAll()
{
    for (int i = 0; i < mNextSlot; i++) {
        mEmployees[i].display();
    }
}

void Database::displayCurrent()
{
    for (int i = 0; i < mNextSlot; i++) {
        if (mEmployees[i].getIsHired()) {

```



```
        mEmployees[i].display();
    }
}

void Database::displayFormer()
{
    for (int i = 0; i < mNextSlot; i++) {
        if (!mEmployees[i].getIsHired()) {
            mEmployees[i].display();
        }
    }
}
}
```

上述各个显示方法都使用了类似的算法。这些方法都会循环处理所有非空的员工记录，如果员工满足显示条件，就会告诉这些员工在控制台上自行显示。

DatabaseTest.cpp

下面是对数据库基本功能的一个简单测试：

```
// DatabaseTest.cpp

#include <iostream>

#include "Database.h"

using namespace std;
using namespace Records;

int main(int argc, char** argv)
{
    Database myDB;

    Employee& emp1 = myDB.addEmployee("Greg", "Wallis");
    emp1.fire();

    Employee& emp2 = myDB.addEmployee("Scott", "Kleper");
    emp2.setSalary(100000);

    Employee& emp3 = myDB.addEmployee("Nick", "Solter");
    emp3.setSalary(10000);
    emp3.promote();

    cout << "all employees: " << endl;
    cout << endl;
    myDB.displayAll();

    cout << endl;
    cout << "current employees: " << endl;
    cout << endl;
    myDB.displayCurrent();

    cout << endl;
    cout << "former employees: " << endl;
    cout << endl;
    myDB.displayFormer();
}
```

#### 1.4.4 用户界面

程序的最后一部分是基于菜单的用户界面，通过这个用户界面可以方便用户使用员工数据库。

UserInterface.cpp

```
// UserInterface.cpp

#include <iostream>
#include <stdexcept>

#include "Database.h"

using namespace std;
using namespace Records;

int displayMenu();
void doHire(Database& inDB);
void doFire(Database& inDB);
void doPromote(Database& inDB);
void doDemote(Database& inDB);

int main(int argc, char** argv)
{
    Database employeeDB;
    bool done = false;

    while (!done) {
        int selection = displayMenu();

        switch (selection) {
            case 1:
                doHire(employeeDB);
                break;
            case 2:
                doFire(employeeDB);
                break;
            case 3:
                doPromote(employeeDB);
                break;
            case 4:
                employeeDB.displayAll();
                break;
            case 5:
                employeeDB.displayCurrent();
                break;
            case 6:
                employeeDB.displayFormer();
                break;
            case 0:
                done = true;
                break;
            default:
                cerr << "Unknown command." << endl;
        }
    }
}
```

```
    return 0;  
}
```

main()函数是一个循环,在此会显示菜单,完成所选择的动作,然后再把这些工作重来一次。对于大多数动作,都已经定义了单独的函数。对于比较简单的动作,比如显示员工,具体代码则放在适当的case子句中。

```
int displayMenu()  
{  
    int selection;  
  
    cout << endl;  
    cout << "Employee Database" << endl;  
    cout << "-----" << endl;  
    cout << "1) Hire a new employee" << endl;  
    cout << "2) Fire an employee" << endl;  
    cout << "3) Promote an employee" << endl;  
    cout << "4) List all employees" << endl;  
    cout << "5) List all current employees" << endl;  
    cout << "6) List all previous employees" << endl;  
    cout << "0) Quit" << endl;  
    cout << endl;  
    cout << "----> ";  
  
    cin >> selection;  
  
    return selection;  
}
```

displayMenu()函数只是输出菜单,并从用户那里得到输入。需要指出的一点,此代码认为用户不会提供超乎寻常的输入,也就是说,如果要求提供一个数字,用户就会输入一个数字(而非其他类型,如字符串)。在读到第14章有关I/O的内容时,你将了解到如何做出防范以避免不好的输入。

```
void doHire(Database& inDB)  
{  
    string firstName;  
    string lastName;  
  
    cout << "First name? ";  
    cin >> firstName;  
    cout << "Last name? ";  
    cin >> lastName;  
  
    try {  
        inDB.addEmployee(firstName, lastName);  
    } catch (std::exception ex) {  
        cerr << "Unable to add new employee!" << endl;  
    }  
}
```

doHire()函数只是获得用户提供的新员工名,并告诉数据库增加此员工。如果出现错误,这个方法会输出一个消息并继续执行,从而妥善地处理错误。



```
void doFire(Database& inDB)
{
    int employeeNumber;
    cout << "Employee number? ";
    cin >> employeeNumber;
    try {
        Employee& emp = inDB.getEmployee(employeeNumber);
        emp.fire();
        cout << "Employee " << employeeNumber << " has been terminated." << endl;
    } catch (std::exception ex) {
        cerr << "Unable to terminate employee!" << endl;
    }
}

void doPromote(Database& inDB)
{
    int employeeNumber;
    int raiseAmount;

    cout << "Employee number? ";
    cin >> employeeNumber;

    cout << "How much of a raise? ";
    cin >> raiseAmount;
    try {
        Employee& emp = inDB.getEmployee(employeeNumber);
        emp.promote(raiseAmount);
    } catch (std::exception ex) {
        cerr << "Unable to promote employee!" << endl;
    }
}
```

doFire()和 doPromote() 都根据一个员工号向数据库请求相应的员工, 然后使用 Employee 对象的公共方法来做修改。

#### 1.4.5 对程序的评价

前面的程序涉及了许多主题, 从非常简单的内容到比较难懂的概念都有所涵盖。还可以采用多种方法扩展这个程序。例如, 这个用户界面并没有提供 Database 或 Employee 类的所有功能。你完全可以修改用户界面 (UI), 加入这些功能。还可以修改 Database 类, 将已经解雇的员工从 mEmployees 数组中删除, 这样就能节省一定的空间。

如果对这个程序的某些部分还不清楚, 可以参考本章前几小节来了解有关的内容。如果还有搞不懂的地方, 最好的学习方法就是实际把这个代码用起来, 并做一些尝试。例如, 如果你不太明确如何使用三元操作符, 可以编写一个简短的 main() 函数, 在这个函数中试一试三元操作符如何使用。

#### 1.5 小结

既然已经对 C++ 的基础知识有所了解, 你可能已经摩拳擦掌地想要成为一位专业 C++ 程序员了。后面的 5 章会介绍一些重要的设计概念。我们将从一个高的层次介绍设计, 不会过分陷入具体的代码, 通过这种方式, 你能对怎样算是好的程序设计有一个更好的认识, 而不至于过早地纠缠于语法细节当中。

在更深入地探究 C++ 语言时, 可以再回过头来参考本章的介绍, 温习一下有关的部分。你可能只需要回顾本章中的一些示例代码, 就能恢复记忆, 把从前忘记的某个概念想起来。

## 第2章 设计专业的C++程序

在编写应用程序之前，先不要具体写任何代码，而应当首先设计程序。你要使用什么数据结构？要编写什么类？如果你们是一个团队，要共同开发程序，先做出这样一个计划就显得极其重要。如果你对同事一无所知，不知道还有哪些人在一同开发同一个程序，或者不清楚谁打算与你共同开发一个程序，就此坐下来直接上手编写程序的话，其后果可想而知！在本章中，我们将教你如何使用专业的C++方法来完成C++设计。

尽管设计很重要，但这也是软件工程中最容易遭到误解和误用的一个方面。许多程序员都可能直接陷入到应用程序中，而在此之前并没有一个清晰的计划，他们是在编写代码的过程中进行设计的，这种情况屡见不鲜。如果采用这种方法，无疑会招致混乱不堪的设计，而且这样的设计往往也极为复杂。基于这种设计，开发、调试和维护工作都会更加困难。尽管不那么直截了当，但是磨刀不误砍柴工，在项目之初多花一些时间来适当地做出设计，将会大大节省整个项目生命期的时间。

第1章已经就C++的语法和特性集上了一堂复习课。第7章还会回过头来介绍C++语法的具体细节，不过，第1部分的余下各章不打算再讨论C++语法，而主要强调编程设计。

读完本章后，你会了解：

- 编程设计的定义
- 编程设计的重要性
- C++所特有的一些设计问题
- 实现有效C++设计的两种基本原则：抽象和重用
- C++程序设计中的具体组件

### 2.1 什么是编程设计

程序设计（program design）或软件设计（software design）就是程序体系结构的规范，要实现这个规范来满足程序的功能和性能需求。所谓设计，就是你打算如何编写程序。一般需要以一份设计文档的形式完成设计。尽管每家公司或每个项目都有自己特定的设计文档格式，但大多数设计文档的一般布局都是一样的，其中包括两个主要部分：

1. 将程序按部分划分为多个子系统，包括子系统间的接口和依赖关系、子系统间的数据流、在各子系统之间来回的输入和输出，以及总的线程模型。
2. 各个子系统的具体细节，包括进一步细分的类、类层次体系、数据结构、算法、特定的线程模型和错误处理细节。

设计文档中通常包含有一些图表，用以显示子系统交互和类层次体系。设计文档的具体格式并不太重要，重要的是设计的考虑过程。

设计的关键在于，要在写程序之前考虑程序。

一般应当在开始着手编写代码之前就完成设计。设计应当提供程序的一个“路线图”，这样一般的程序员都能按照这个路线图来实现应用。当然，一旦开始编写代码，设计总免不了需要修改，而且总会遇到一些原先未曾想到的问题。通过软件工程过程，应该能够提供足够的灵活性来应对这样一些改变。第 6 章将更为详细地介绍几种不同的软件工程过程模型。

## 2.2 编程设计的重要性

为了能尽早地开始编写程序，人们往往想要跳过设计这一步，或者只是草率地匆匆完成设计。好像只有编译代码和运行代码才能有成就感，觉得自己有所长进。如果你已经或多或少地知道了想要怎样建立程序的结构，在这种情况下还来完成正式的设计似乎只是浪费时间。另外，与编写代码相比，撰写设计文档可没有那么有意思。如果只想整天写文档，那你肯定做不了真正的计算机程序员！作为程序员，我们自己也很清楚这种直接上手编写代码的诱惑有多大，不仅如此，我们偶尔也确实会陷入其中。不过，除了最简单的项目外，这样仓促上阵无疑会带来问题。

为了帮助理解编程设计的重要性，我们可以做一个实际的对比。假设你有一块土地，想在上面盖一座房子。建筑工人开始动工的时候，你告诉他想看看设计蓝图，“什么蓝图？”他这样回答，“我当然知道自己在做什么。我可不想提前为每个小细节都做出规划。你想盖两层的房子？没问题，几个月前我刚盖过一个一层的房子，我可以按着那个模型动手。”

尽管对他不是很相信，但你还是克制住自己，容他继续开工。几个月之后，你注意到在房子外面出现了一些管线，而正常情况下，应该是在墙内走线才对。对于这种不正常的情况，你会质询建筑工人，他可能会这样回答，“哦，我忘了在墙上为管线留出空间了。刚想出这种全新的净墙（drywall）技术时，真是让我兴奋不已。不过，即使管线在外面也无所谓，它还是能正常工作的，而且功能才是第一位的。”你开始对他的方法存在置疑了，不过，尽管你的判断正确，但是没有采取任何行动，而仍然允许他继续工作。

房子完工后，第一次参观房子时，你注意到厨房居然没有水池。建筑工人可能解释说“我们完成了厨房 2/3 的工作时才发现没有为水池留出空间。我们不想重新返工，可以在隔壁增设一个单独的水池间，这样也不错，对不对？”

编写程序而没有一个设计，就像是盖房子没有蓝图一样。

如果把这种情况搬到软件领域中来，对于建筑工人的诸项解释，听上去是不是很熟悉呢？你自己是不是也有这样的经历，就像把管线置于房子之外一样，你是不是也做过此类“丑陋”的解决方案呢？举例来说，对于由多个线程共享的队列数据结构，你可能忘记了要在其中设置加锁机制。等到意识到存在这个问题时，好像更容易的做法是要求所有线程都记住自行完成加锁。你可能会说，这么做确实不算“漂亮”，但起码能工作。不过，以后有新的成员加入到项目中来，他可能认为数据结构内置有加锁机制，但由于事实并非如此，最后将无法保证对共享数据的互斥访问，这就会导致一个竞争条件 bug，可能需要花费 3 周的时间才能找出问题所在。直到此时，你才会发现原先的看法是不妥当的。

在编写代码之前建立一个正式的设计，这有助于确定如何将所有东西加以结合。就像房屋蓝图可以显示出房间之间的相互关系，以及各个房间如何共同满足房屋的需求一样，程序的设计也可以显示出程序中子系统间的相互关系，以及各个子系统如何协作来满足软件需求。如果没有设计计划，很可能会遗漏掉子系统间的连接，原本可以重用或共享的信息未能真正重用或共享，另外可能会错过一些完成任务的捷径。如果没有获得设计所给予的“整体图”，就会过分沉溺于单个的实现细节中，而忘记了支持架构和所要实现的目标才是重点。不仅如此，设计还能向项目的所有成员提供成文的文档，以供参考。

如果上述对照分析还不足以说服你先设计再编写代码，下面再提供一个例子，由此可以看出，如果

直接陷入代码编写，就无法得到一个最优的设计。假设你希望编写一个象棋程序，但在开始编程之前并没有对整个程序进行设计，而是决定直接完成最简单的部分，然后再慢慢地实现更困难的部分。根据第1章介绍的面向对象观点（在第3章还将更详细地介绍），你决定用类来建立棋子的模型。卒是最简单的棋子，你准备由此开始。考虑了卒的属性和行为之后，你写出了——一个类，其属性和行为如表2-1所示。

表 2-1

类	属 性	行 为
卒	在棋盘上的位置 颜色（黑或白） 是否被吃掉	移动 检查移动是否合法 绘制 提升（走到棋盘对岸时）

当然，你并没有写这样一个表，而是直接开始实现的。你很高兴地转而开始实现下一个最简单的棋子：相。考虑了它的属性和功能之后，你又写了一个类，其属性和行为如表2-2所示。

表 2-2

类	属 性	行 为
相	在棋盘上的位置 颜色（黑或白） 是否被吃掉	移动 检查移动是否合法 绘制

同样地，因为你直接进入了编写代码阶段，也没有生成这样一个表。不过，此时你开始有些怀疑了，是不是哪些地方出了问题。相和卒看上去很像。实际上，它们的属性完全相同，许多行为也是一样的。尽管卒和相在移动这个行为上的具体实现有所不同，但它们都需要能够移动，这一点是不争的事实。如果在投入编写代码之前先对程序做了设计，可能早就能意识到不同的棋子实际上是很相似的，而且会得出某种方法，使这些公共功能只编写一次。第3章将介绍有关面向对象设计的技术。

另外，棋子的不同方面还取决于程序的其他子系统。例如，如果不知道怎么对棋盘建模，就无法具体地表示棋子在棋盘上的位置。另一方面，也许你想这样设计程序：让棋盘以一种特定的方式管理棋子，使得棋子无需知道自己的位置。无论哪一种情况，设计棋盘之前先在棋子类中加入位置这个属性都会带来问题。再举一个例子，如果还没有确定程序的用户界面，怎么编写一个棋子的绘制方法呢？这个绘制是基于图形的还是基于文本的？棋盘是什么样子的？这里的问题在于，程序的子系统并不是独立存在的，它们要与其他子系统相互关联。设计所做的大部分工作就是要确定和设计这些关系。

## 2.3 C++ 设计有什么不同之处

与其他语言的设计相比，C++语言的许多方面都使得C++设计有所不同，而且更为复杂。

- 首先，C++有一个丰富的特性集。这几乎就是C语言的一个完备超集，另外还要加上类和对象、操作符重载、异常、模板和其他一些特性。由于这个语言的规模如此之大，就使得设计成为一项很棘手的任务。
- 其次，C++是一种面向对象语言。这说明，设计应当包含类层次体系、类接口和对象交互。这种设计与C或其他过程性语言的“传统”设计大相径庭。第3章将重点介绍C++中的面向对象设计。
- C++还有一个独特的方面，这就是它提供了大量工具来设计通用的和可重用的代码。除了基本的类和继承，还可以使用其他一些语言工具来完成有效的设计，如模板和操作符重载。第5章将介

绍有关可重用代码的设计技术。

- 另外，C++ 提供了一个有用的标准库，其中包括一个字符串类、一些 I/O 工具，以及许多常用的数据结构和算法。而且，许多设计模式或解决问题的常用方法同样适用于 C++。第 4 章将涉及如何使用标准库进行设计，还将介绍设计模式。

正是由于上述问题的存在，完成一个 C++ 程序的设计可能相当困难。本书作者之一就曾经花了数天之久在纸上先勾画出设计思想，再划掉，又加上一些思想，再把它们划掉，如此往复地完成这个过程。有时这个过程是大有裨益的，数天（或者数周）之后，你会得到一个简洁、高效的设计。但有时这个过程可能很令人厌烦，而且不会提供任何帮助。你要一直都很清楚自己是否真的有进步，这一点很重要。如果发现自己陷入了困境，可以采取以下某个行动：

- 寻求帮助。咨询同事、导师，或者参考相关的书、新闻组或上网查看网页。
- 与别人合作一段时间。以后再回过头来做出设计选择。
- 做出决定，并继续。尽管某个决定不能算理想方案，但先做这个选择，尝试看能否成行。如果是一个不正确的选择，将很快暴露出来。不过，你也可能发现这是一个可以接受的方法。对于想要完成的任务，除了这种设计之外，可能并没有其他的捷径。有时即使是一种“不漂亮”的解决方案，你也必须接受，因为这是惟一能够满足需求的实际策略。

要记住，得到好的设计会很难，要想达到目的需要大量实践。不要期望一夜之间就能变成专家，你可能会发现掌握 C++ 设计比掌握 C++ 编码更为困难，对此请不要感到奇怪。

## 2.4 C++ 设计的两个原则

C++ 中有两个基本设计原则：抽象（abstraction）和重用（reuse）。这些原则如此重要，甚至可以把它们作为这本书的主旨。全书中将反复出现这两个原则，另外在各个领域的有效 C++ 程序设计中也会时常会出现他们的身影。

### 2.4.1 抽象

要理解抽象（abstraction）原则，最简单的方法是通过一个实际对照来说明。电视作为一个简单的技术，在大多数家庭中都已普及。你对电视的功能可能已经很熟悉了：可以打开和关掉电视，可以转换频道，可以调节音量，还可以增加一些外部部件，如喇叭、VCR 和 DVD 机等。不过，你能解释在这个黑盒子里到底是怎么工作的吗？也就是说，你知道它是如何通过大气或者通过线缆接收信号的，又是如何完成信号转换，并在屏幕上显示出来的？我们当然无法将电视如何工作解释清楚，不过对于怎么使用它却可能很精通。这是因为电视很清楚地将其内部实现与外部接口相分离。我们只是通过其接口与电视交互：包括电源开关、频道转换按钮和音量控制按钮等。我们不知道电视的工作原理，对此也不关心，我们不想知道它是怎样使用阴极管或另外某种技术在屏幕上生成图像的。这些并不重要，因为这一切不会对接口产生影响。

#### 抽象带来的好处

在软件领域中，抽象原则也是类似的。可以使用代码，而无需了解其底层实现。先举一个简单的例子，你的程序可能调用了 `sqrt()` 函数（此函数在头文件 `<cmath>` 中声明），而无需知道这个函数具体使用了什么算法来计算平方根。实际上，对于不同版本的库，平方根计算的底层实现可以有所不同，只要接口保持不变，函数调用就仍能正常工作。抽象的原则还可以延伸至类。在第 1 章中曾介绍过，可以使用类 `ostream` 的 `cout` 对象将数据作为流传送至标准输出，如下所示：

```
cout << "This call will display this line of text\n";
```

在以上代码行中，使用了 `cout` 插入（insertion）操作符（带一个字符数组）的公开接口。不过，`cout` 是如何管理从而在用户界面上显示这行文本的呢？对此无需了解。只要知道公共接口就可以了。`cout` 的底层实现完全可以修改，只要保证所提供的行为和接口保持不变即可。第14章将更详细地讨论 I/O 流。

### 在设计中纳入抽象

应当适当地设计函数和类，从而使你和其他程序员可以直接使用这些函数和类，而无需知道（或依赖于）函数和类的底层实现。作为一个将实现暴露在外的设计，与将实现隐藏在接口内的设计相比，要了解它们之间有什么区别，还是来考虑前面的象棋程序。你可能希望用一个二维的 `ChessPiece` 对象指针数组来实现棋盘。可以如下声明和使用这个棋盘：

```
ChessPiece* chessBoard[10][10];  
...  
ChessBoard[0][0] = new Rook();
```

不过，这种方法没有用到抽象的概念。每个程序员要想使用这个棋盘，都会知道它实现为一个二维数组。要想对这个实现做某种修改，例如修改为一个向量数组，将会很困难，这是因为需要把整个程序中用到棋盘的地方都加以修改。在此接口与实现就未做分离。

还有一个更好的办法，就是将棋盘建模为一个类。这样就可以提供一个接口，而该接口隐藏了底层的实现细节。以下是一个 `ChessBoard` 类的例子：

```
Class ChessBoard {  
public:  
    // This example omits constructors, destructors, and the assignment operator.  
    void setPieceAt(ChessPiece* piece, int x, int y);  
    ChessPiece& getPieceAt(int x, int y);  
    bool isEmpty(int x, int y);  
protected:  
    // This example omits data members.  
};
```

注意，这个接口没有明确指定任何底层实现。`ChessBoard` 可以是一个二维数组，不过接口对此并未做严格要求。修改实现时无需修改接口。另外，实现还可以提供额外的功能，如越界检查，对于这一点前面的第一种方法是无能为力的。

通过这个例子，希望你能了解到抽象是 C++ 编程中的一项重要技术。第3章和第5章将更为详细地介绍抽象和面向对象设计，另外第8章和第9章将完备地提供有关的详细内容，介绍如何编写自己的类。

### 2.4.2 重用

C++ 的第二个基本设计原则是重用（reuse）。同样地，通过做一个实际的对照分析将有助于理解这个概念。假设你放弃编程，成为了一个面包师。在第一天的工作中，大师傅让你烤些饼干。为了达到他的要求，你在烹饪书里找到了巧克力饼干的食谱，接下来把配料混合在一起，在饼干模子上做出饼干，然后把饼干模子放到烤箱里烘烤。看到烤出来的饼干，大师傅可能会很满意。

下面我们要指出一些显而易见的情况，专门把它们指出来似乎有些小题大做，因为在我们看来，这实在再显然不过了。在此你没有自己造炉子来烤饼干，也没有自己搅奶油、自己磨面、自己做巧克力片。你肯定会说，“那是当然的了”。如果你是一个真正的厨师，这一点毋庸置疑，但是如果你是一个程序员，要编写一个烤面包模拟游戏，又会怎样呢？倘若如此，你可能只好考虑自行编写程序的每个组件，从巧



克力片到炉子都要亲力而为。不过，你可以看看周围有没有可以重用的代码来节省时间。也许你的某个同事写过一个烹饪模拟游戏，其中有一些很不错的炉子代码。尽管这些代码不完全满足要求，没有完成你需要的每一件工作，不过可以对它进行修改，并增加必要的功能。

还有一样东西是直接拿来用的，你在这里参考了一个食谱，而不是自己研究出来一个食谱。同样的，这当然也无需多说。不过，在 C++ 编程中可并不像这么显而易见。尽管 C++ 中有许多标准方法可以用来解决一些反复出现的问题，但是许多程序员还是很固执，每次设计时都会把这些策略重新再设计一次。

### 重用代码

使用既有代码，这个思想对你来说可能并不陌生。从用 `cout` 进行打印的那一天起你就在重用代码。你并没有编写具体的代码在屏幕上打印数据，而是使用了既有的 `ostream` 实现来完成这个工作。

遗憾的是，程序员通常并没有充分利用所有可用的代码。你的设计应当考虑到既有的代码，并在适当的时候加以重用。

例如，假想编写一个操作系统调度器。这个调度器是操作系统的一个组件，负责确定哪个进程运行，以及要运行多久。因为你想实现一种基于优先级的调度，所以可能意识到需要一个优先队列，并在这个队列中存储等待运行的进程。对于这种设计，一种直接的方法就是编写自己的优先队列。不过，你应该知道，C++ 标准模板库（standard template library，STL）已经提供了一个 `priority_queue` 容器，可以使用这个容器存储任何类型的对象。因此，应当在调度器的设计中加入 STL 中的这个 `priority_queue`，而不是重新编写自己的优先队列。第 4 章将更为详细地说明代码重用，还将介绍标准模板库。

### 编写可重用代码

重用设计原则不仅适用于你编写的代码，还适用于你使用的代码。应当适当地编写程序，从而可以重用类、算法和数据结构。你和你的同事不仅能够当前项目中利用这些组件，还能在将来的项目中使用这些可重用的代码。总的说来，应当避免设计过于特定的代码（仅适用于当前情况）。

C++ 中提供了一个编写通用代码的技术，这就是模板。以下的例子显示了一个模板化的数据结构。如果你以前从未见过这个语法，也不必担心！第 11 章将深入地介绍这个语法。

这里没有编写一个保存 `ChessPiece` 的特定 `ChessBoard` 类（如前所示），而是考虑编写了一个通用的 `GameBoard` 模板，这个模板可以用于任何类型的二维棋类游戏，如象棋或跳棋。只需修改类声明，使之将所要存储的棋子作为一个模板参数，而不是将棋子硬编码（直接编写）到接口中。此模板如下所示：

```
template <typename PieceType>
class GameBoard {
public:
    // This example omits constructors, destructors, and the assignment operator.
    void setPieceAt(PieceType* piece, int x, int y);
    PieceType& getPieceAt(int x, int y);
    bool isEmpty(int x, int y);
protected:
    // This example omits data members.
};
```

通过对接口做这样一个简单的修改，就有了一个通用的游戏棋盘类，它可以用于任何二维棋类游戏。尽管这里的代码修改很简单，但是在设计阶段做出这样一些决策是非常重要的，这样一来，就能有效而且高效率地实现代码了。

### 重用思想

如前面的烤饼干例子所示,如果每做一道菜(每烤一种饼干)都自造一个食谱,这是很可笑的。不过,程序员通常会在设计中犯这种错误。他们可能没有利用既有的“食谱”(或模式)来设计程序,而是每设计一个程序都重新实现这些技术。在各种不同的 C++ 应用中已经出现了许多设计模式(design pattern)。作为一个 C++ 程序员,你应当熟悉这样一些模式,并在程序设计中有效地结合这些模式。

例如,你可能希望如下设计象棋程序,使之有惟一的一个 ErrorLogger 对象,由它将不同组件的所有错误串行化记录到一个日志文件中。在尝试设计 ErrorLogger 类时,你意识到,在一个程序中从 ErrorLogger 类实例化多个对象可能是灾难性的。你可能还希望能够从程序的任何位置访问这个 ErrorLogger 对象。在 C++ 程序中,经常会出现这种需求,即要求有惟一的、可全局访问的类实例,而且为实现这样的实例已经有了一种标准策略,这称为单例模式(singleton)。因此,此时如果打算使用单例模式,这就是一个好的设计。第 5 章、第 25 章和第 26 章将更详细地介绍设计模式和技术。

## 2.5 设计一个象棋程序

这一节将介绍如何采用一种系统的方法来设计一个 C++ 程序,在此以一个简单的象棋游戏应用为背景。为了提供一个完备的例子,这里的某些步骤会涉及后面章节中才会谈到的概念。你可以现在看这个例子,对设计过程有一个整体的认识,不过等看完第 1 部分之后,可能还希望再把这个例子回顾一遍。

### 2.5.1 需求

着手设计之前,有一点很重要,这就是要准确地把握程序的功能需求和性能需求。理想情况下,这些需求应当以需求规范的形式明确地给出。棋盘需求包含以下各类规范,不过实际的需求可能会更多,更为详细:

- 程序要支持象棋的标准规则(下法)。
- 程序要支持两个玩家(真正的人)。这个程序不支持人工智能机器玩家。
- 程序要支持一个基于文本的界面:
  - 程序要以 ASCII 文本方式显示棋盘和棋子。
  - 玩家要输入数字(表示棋盘上的位置)来表示如何走棋。

基于上述需求,可以确保你对程序做了正确的设计,从而得到用户所期望的表现。如果用户只需要一个基于文本的界面,你可能不会希望花时间为这个象棋游戏设计和编写一个图形用户界面。反过来,如果用户更喜欢图形用户界面,你也必须有所了解,这一点很重要,这样就不至于将程序设计为仅有基于文本的界面,而把基于图形的可能性完全排除在外。

### 2.5.2 设计步骤

应当采用一种系统的方法来设计程序,即从一般到特殊。以下步骤并不适用于所有程序,不过,这些步骤确实提供了一个一般原则。设计应当适当地包括一些图表。这个例子就包括有一些示例图表。可以遵循这里所用的格式,也可以建立自己的格式。

在画软件设计图时,没有孰对孰错之分,只要设计图清晰,你和你的同事能够了解其含义即可。

#### 将程序划分为子系统

第一步是把程序划分为通用功能子系统,并明确子系统间的接口和交互。此时,不必操心数据结构和算法(甚至类)的特定细节,只需对程序的各个部分及其交互有一个总的认识就可以了。可以把子系统列在一个表中,在这个表中表示出子系统的高层行为或功能、子系统向其他子系统提供的接口,以及

此子系统使用了其他子系统的哪些接口。象棋游戏子系统的表可能如表 2-3 所示。

表 2-3

子系统名	个数	功能	所提供的接口	所使用的接口
GamePlay	1	开始游戏 控制游戏流 控制绘制 宣布赢家 结束游戏	Game Over (游戏结束)	Take Turn (Player) Draw (Chess Board)
Chess Board	1	保存棋子 检查是否平局和是否有 输赢 自行绘制	Get Piece At (获得棋子) Set Piece At (设置棋子) Draw (绘制)	Game Over (Game-Play) Draw (Chess Piece)
Chess Piece	32	自行绘制 自行移动 检查移动是否合法	Draw (绘制) Move (移动) Check Move (检查移动)	Get Piece At (Game Board) Set Piece At (Game Board)
Player	2	与用户交互：提示用户走 棋、得到用户走棋信息 移动棋子	Take Turn (轮流下棋)	Get Piece At (Game Board) Move (Chess Piece) Check Move (Chess Piece)
ErrorLogger	1	将错误消息写至日志文件	Log Error (记录错误)	无

如表 2-3 所示，这个象棋游戏的功能子系统包括 1 个 GamePlay 子系统、1 个 ChessBoard、32 个 ChessPiece、2 个 Player，以及 1 个 ErrorLogger。不过，这并不是实现象棋游戏惟一可行的方法。在软件设计中，与编程一样，为达到同一个目标通常都有多种不同的方法。并非所有方法都一样，有些方法肯定要优于另外一些方法。不过，一般都会有多种同样有效的方法。

通过适当地划分子系统，可以将程序分解为基本功能部件。例如，在象棋游戏中，Player 就是一个区别于 Class Board、Chess Piece 或 GamePlay 的子系统。把玩家规到 GamePlay 对象中是没有意义的，因为它们在逻辑上是完全分离的子系统。其他的选择则没有这么明显。增加一个单独的用户界面子系统是否可行？如果你想提供不同类型的用户界面，或者希望以后能轻松地修改用户界面，可能会把这方面的内容分离到一个单独的子系统中。要做出此类选择，不仅需要考虑程序当前要达到的目标，还要考虑将来可能会有哪些目标。

因为从表中通常很难看出子系统间的关系，用图的形式显示程序中的子系统往往很有帮助，如图 2-1 所示。在这个图中，箭头表示从一个子系统到另一个子系统的调用（为简单起见，在此省略了 Error Logger 子系统）。

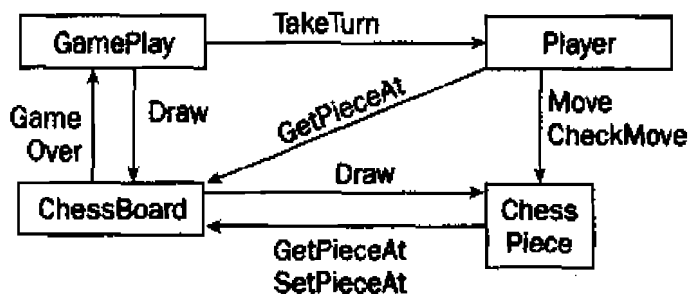


图 2-1

### 选择线程模型

在这一步中,要选择程序中使用多少个线程,并明确线程的交互。还要为共享数据指定加锁机制。如果你对多线程程序不熟悉,或者你的平台不支持多线程,就应该建立单线程程序。不过,如果程序有多项不同的任务,每个任务都应当并行进行,那么多线程则是一个不错的选择。例如,图形用户界面应用通常就有一个线程专门完成主要应用工作,而另一个线程用于等待用户按键或选择菜单项。

由于线程与具体平台有关,这本书不打算讨论多线程编程。第18章将讨论使用 C++ 时有关平台的一些考虑。

这个象棋程序只需要一个线程来控制游戏流。

### 为每个子系统指定类层次体系

在这一步中,要确定程序中要编写怎样的类层次体系。象棋程序只需要一个类层次体系,用以表示各种棋子。这个层次体系可能如图 2-2 所示。

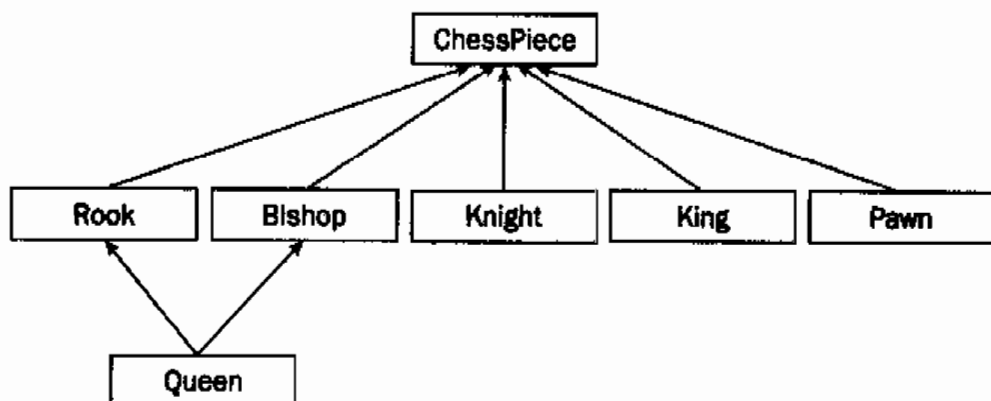


图 2-2

在这个层次体系中,有一个通用的 ChessPiece 类,这个类将用作为超类。以上层次体系使用了多重继承来显示王后实际上是车和相的结合。

第3章将解释设计类和类层次体系的详细内容。

### 为每个子系统指定类、数据结构、算法和模式

这一步要考虑更深层次的细节,并指定各个子系统的一些特定的内容,其中包括为每个子系统编写的特定类。最后很可能发现,你会为每个子系统本身建立一个类。有关信息还是用表 2-4 来总结。

表 2-4

子系统名	类	数据结构	算 法	模 式
GamePlay	GamePlay 类	GamePlay 对象包括一个 ChessBoard 和两个 Player 对象	简单循环,让每个玩家轮流走棋	无
Chess Board	ChessBoard 类	ChessBoard 对象是一个二维的 ChessPiece 数组, ChessBoard 存储有 32 个 ChessPiece	每走一步,检查是否有输赢(或平局)	无
Chess Piece	ChessPiece 抽象超类, Rook、Bishop、Knight、King、Pawn 和 Queen 类	每个棋子会存储它在棋盘上的位置	棋子向棋盘询问其他位置上的棋子,检查走得是否合法	无

(续)

子系统名	类	数据结构	算法	模式
Player	一个 Player 类	两个玩家对象 (黑和白)	轮流算法: 循环提示用户走棋, 检查走得是否合法, 并移动棋子	无
ErrorLogger	一个 ErrorLogger 类	要记录的消息队列	将消息放入缓冲区, 并周期性地写入一个日志文件	单例模式, 用以确保只有一个 ErrorLogger 对象

正式设计文档的这一部分通常还会表示每个类的具体接口, 不过, 这个例子没有做到那么细。

设计类、选择数据结构、算法和模式, 这些工作可能很复杂。要时刻谨记本章前面讨论的抽象和重用这两个原则。对于抽象, 关键是要分别考虑接口和实现。首先, 要从用户角度指定接口, 确定希望组件做些什么。再通过选择数据结构和算法确定组件如何达到目的。对于重用, 要先熟悉标准的数据结构、算法和模式。另外, 还要确保自己了解 C++ 中的标准库代码, 以及可供使用的任何专用代码。

第 3 章、第 4 章和第 5 章将更为详细地讨论这些问题。

#### 为每个子系统指定错误处理

在这个设计步骤中, 要明确每个子系统中的错误处理。错误处理应当包括系统错误 (如内存分配失败) 和用户错误 (如非法输入) 的处理。要指定各个子系统是否使用异常。下面还是通过表 2-5 来总结有关的信息。

表 2-5

子系统名	处理系统错误	处理用户错误
GamePlay	如果无法为 ChessBoard 或 Player 分配内存, 用 ErrorLogger 记录一个错误, 并终止程序	无 (没有直接的用户界面)
Chess Board	如果无法为其自身或 ChessPiece 分配内存, 用 ErrorLogger 记录一个错误, 并抛出一个异常	无 (没有直接的用户界面)
Chess Piece	如果无法分配内存, 则用 ErrorLogger 记录一个错误, 并抛出一个异常	无 (没有直接的用户界面)
Player	如果无法分配内存, 则用 ErrorLogger 记录一个错误, 并抛出一个异常	检查用户输入的走法是否正常, 确保不至于走到棋盘外面去; 倘若真的会走到棋盘外面去, 提示用户重新输入 在真正移动棋子之前先检查各步是否合法; 如果不合法, 提示用户再考虑其他走法
ErrorLogger	尝试记录一个错误, 如果无法分配内存, 则终止程序	无 (没有直接的用户界面)

错误控制的一般原则是一切都要处理。要仔细周全地考虑到所有可能的错误条件。如果漏掉了一种可能性, 最后它就会在程序中作为一个 bug 出现在你面前! 不要让问题变成“未预料到的”错误才暴露出来。要考虑到所有可能性: 内存分配失败, 非法的用户输入, 磁盘故障, 网络故障等等。不过, 从针对象棋游戏的这个表可以看到, 对用户错误的处理应当与内部错误的处理有所区别。例如, 用户输入了一个非法的走法时, 不应导致象棋程序终止。

第 15 章将更深入地讨论错误处理。

## 2.6 小结

本章介绍了专业 C++ 设计方法。我们希望，你能由此了解到软件设计的的确确是所有编程项目中首要的一步。在此，你还了解了 C++ 的哪些方面导致了设计困难，这包括 C++ 的面向对象性，丰富的特性集和标准库，以及编写通用代码的诸多工具。有了以上基础，你应该有更充分的准备，可以着手进行 C++ 设计了。

本章介绍了两个设计原则：抽象和重用。所谓抽象，就是将接口与实现相分离，这个概念贯穿本书始终，应当作为所有设计工作的指导原则。重用的思想（无论是代码重用还是思想重用）在实际的项目经常会出现，在本书中也会屡屡看到它的身影。应当重用既有的代码和思想，并尽可能地将代码编写为可以重用。

既然已经了解了设计的重要性，也知道了基本的设计原则，下面可以开始学习第 1 部分的余下内容了。第 3 章将介绍在设计中利用 C++ 的面向对象方面时有哪些策略。第 4 章和第 5 章对重用既有代码和思想以及为编写可重用的代码提供了相应的原则。第 1 部分的最后是第 6 章，其中讨论了软件工程模型和过程。



## 第3章 基于对象的设计

通过第2章的学习，你已经对好的软件设计有了一定的认识，与好的设计相辅相成的还有对象思想，下面就要介绍有关对象的概念。只是在代码中使用对象的程序员与真正掌握面向对象程序设计的程序员之间是有差别的，区别就在于他们的对象以何种方式彼此相关，以及对象与程序的总体设计之间存在怎样的关联。

本章先从过程性程序设计过渡到面向对象程序设计。也许你用对象已经很多年了，即便如此，可能还希望通过阅读本章，对如何考虑对象获得一些全新的思想。在讨论对象之间的各种不同关系时，我们还会指出程序员在构建面向对象程序时常常遭遇的陷阱。你还将了解到抽象原则与对象有什么关系。

### 3.1 面向对象的世界观

从过程性（C风格）代码设计过渡到面向对象代码设计时，要记住最重要的一点：面向对象程序设计（object-oriented programming, OOP）只是以另一种思路来考虑程序中发生了什么。程序员在充分理解对象是什么之前，通常会先陷到 OOP 的新语法和新术语中不能自拔，这种情况屡见不鲜。本章只对编写代码做简单的介绍，而把重点放在概念和思想上。有关 C++ 对象的具体语法，请参见第 8 章、第 9 章和第 10 章。

#### 3.1.1 我是在以过程性思维思考吗

过程性语言（如 C）把代码划分为小的代码块，（理想情况下）每个代码块都完成一个任务。如果没有 C 中的过程，所有代码都会堆到 `main()` 函数里。这样的代码很难阅读，你的同事也会对此大加光火，而且这还算是好的，这种代码可能还存在更严重的危害。

所有代码是否都放在 `main()` 中，或者是否划分到较小规模的代码块中（并有描述性命名和注释），计算机对此并不关心。过程是一种抽象，这种抽象能为作为程序员以及阅读和维护代码的人提供帮助。这个概念围绕着有关程序的一个基本问题展开，即这个程序要做什么？如果直接回答这个问题（比如用英语），你就是在用过程性思维考虑问题。例如，通过回答这个问题，你可能开始设计一个股票选择程序：首先，程序从 Internet 上得到股票的股价。然后，基于特定的标准对此数据进行排序。接下来，对已排序的数据完成分析。最后，输出一组股票买入和卖出建议信息。开始编写代码时，你可能会把心里想的这个模型直接转换成一系列 C 函数：`retrieveQuotes()`、`sortQuotes()`、`analyzeQuotes()` 和 `outputRecommendations()`。

虽然 C 把过程称为“函数”，但是 C 并不是函数式语言。“函数式”（functional）一词与过程性恰好相反，它是指诸如 Lisp 之类的语言，这些函数式语言使用了一种完全不同的抽象。

如果你的程序会严格地遵循特定的步骤运作，过程性方法往往就很合适。不过，在当前的大型应用中，事件很少会作为一个线性序列按顺序发生。通常用户可以在任何时刻完成任何命令。过程性思维对

于数据如何表示也未做任何考虑。在前面的例子中，并没有讨论股票股价到底是什么。

如果你正是基于这样一种过程性思维方式来编写程序，也不必担心。一旦你认识到 OOP 只是另外一种思维方式，即以一种更灵活的思路来考虑软件，就会很自然地采纳这种思想。

### 3.1.2 面向对象思想

过程性方法的基础是问这样一个问题：“这个程序要做什么？”，与此不同，面向对象方法则问了另一个问题：“我要为哪些实际对象建模？”。OOP 所基于的思想是：不应将程序划分为任务，而应当划分为物理对象模型。尽管乍看上去有些抽象，不过如果从物理对象的类、组件、属性和行为等方面来考虑，这个概念就会很清楚了。

#### 类

类 (class) 有助于将对象与其定义相区别。可以考虑桔子来作为例子。在谈到一般意义上的桔子时，是指它是生长在树上的一种好吃的水果，这与某个特定的桔子是不同的（比如现在我桌子上的这个桔子，它的汁溅到了我的键盘上）。

在回答“什么是桔子”这个问题时，你说的是称为桔子的这一类东西。所有桔子都是水果。所有桔子都生长在树上。所有桔子都是桔类的一个品种。所有桔子都有某种特定的口味。类就是一个封装，其中封装了定义一类对象的有关内容。

在描述一个特定的桔子时，你所指的是一个对象。所有对象都属于一个特定的类。因为桌子上的对象是一个桔子，我知道它属于桔子类。因此，我能肯定这是一个生长在树上的水果。我还能进一步指出，这是一个一般品种的桔子，口味非常好。对象是类的一个实例 (instance)，这个特定实例的特性会与同一个类的其他实例有所区别。

再来看一个更具体的例子，还是考虑前面的股票选择应用的例子。在 OOP 中，“股价”是一个类，因为它定义了构成股价的抽象概念。特定的股价，如“当前 Microsoft 的股价”就是一个对象，因为这是该类的一个特定实例。

如果你原先有 C 背景，可能认为类和对象与 C 中的类型和变量有相似之处。实际上，在第 8 章中，我们将会看到，类的语法与 C 的 struct 的语法确实很类似。对象的语法与 C 风格变量的语法也相当接近。

#### 组件

如果考虑复杂的实际对象，如飞机，就能很容易地看到，它由多个更小的组件 (component) 所组成。这包括机身、操纵装置、起落装置、发动机和其他一些部件。在过程性程序设计中，把复杂任务分解为较小过程可谓是一个基本环节。与此相仿，将对象考虑为由更小组件所组成，这在 OOP 中同样堪称基石。

组件实际上也是类，只不过更小一些，而且更为特定。好的面向对象程序可能有一个 Airplane 类，不过，如果完全由这个类描述飞机，那么这个类可能很大。与此不同，Airplane 类处理了多个较小的、更可管理的组件。每个组件可能又有自己的子组件。例如，起落装置是飞机的一个组件，轮子又是起落装置的一个组件。

#### 属性

属性 (property) 是不同对象有所区别的部分。再来看前面的 Orange 类，应该记得所有桔子都定义为属于某个品种，而且有一种特定的口味。这两个特性就是属性。所有桔子都有相同的属性，只不过值不同而已。我的桔子的口味“非常好”，你的桔子可能“很难吃”。

还可以在类层次上考虑属性。前面已经提到过，所有桔子都是水果，而且都生长在树上。这些就是这类水果（水果类）的属性，而桔子的特定品种则由特定水果对象确定。类属性由类的所有成员共享，

而对象属性出现在类的所有对象中，但是不同对象的对象属性有不同的值。

在股票选择例子中，股价就有多个对象属性，包括公司名、股票代码、当前价格和其他统计信息。

属性就是描述对象的一些特性。属性回答了这样一个问题：是什么使得这个对象与众不同？

#### 行为

行为 (behavior) 回答了以下问题：“这个对象会做什么？”或者是“我能对这个对象做些什么？”。在桔子这个例子中，它本身做不了多少事情，不过我们可以对它做一些工作。首先，桔子可以吃，这就是一个行为。与属性一样，可以在类层次或对象层次考虑行为。所有桔子都可以用同样的方式吃掉。不过，在其他某种行为上可能会存在差异，如沿着斜坡向下滚，非常圆的桔子和比较扁的桔子相比，在这个行为上就会反映出不同来。

股票选择例子提供了一些更为实际的行为，应该记得，在采用过程性思维考虑时，对于程序要分析股价这一点，我们确定这要作为程序的函数之一。用 OOP 思想来考虑时，我们可能决定股价对象可以自行分析！这样一来，分析就成为了股价对象的一个行为。

在面向对象程序设计中，会从过程中移出大量功能代码，并且转入到对象中去。通过构建有特定行为的对象，并定义对象的交互方式，OOP 提供了一种更强大的机制，可以将代码与它所操作的数据相关联。

#### 汇总

有了上述概念，可以再来审视前面的股票选择程序，并以一种面向对象的方式重新进行设计。

如前所述，由“股价”这个类起步就很合适。不过，为了得到一个股价表，程序需要有一组股价的概念，这通常称为集合 (collection)。因此，更好的设计可能应该有一个表示“股价集合”的类，这个类由表示单个“股价”的较小组件所组成。

再来看属性，集合类至少要有一个属性，即所接收的实际股价表。另外，它可能还有其他的属性，如最近一次接收股价的准确日期和时间，以及所得到的股价个数。至于行为，“股价集合”要能够与一个服务器会话，得到股价，并提供一个有序的股价表。这就是“获取股价”行为。

股价类可以有前面所述的属性：公司名、股票代码、当前价格等等。同样地，前面还指出了，股价类要有一个分析行为。你还可以考虑其他行为，如买入和卖出股票等。

画一些图来表示组件之间的关系往往很有用。图 3-1 用多条线来表示一个“股价集合”包含有多个“股价”对象。

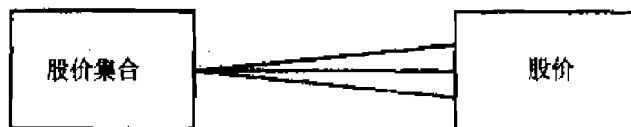


图 3-1

要以直观的方式表示类，还有一种有用的方法，这就是在脑海中建立程序的对象表示，同时用表列出类的属性和行为（如表 3-1 所示）。

表 3-1

类	相关组件	属性	行为
Orange (桔子)	无	颜色 口味	吃 滚 抛

(续)

类	相关组件	属 性	行 为
Collection of Stock Quotes (股价集合)	由单个股价对象组成	单个股价 时间戳 股价个数	获取股价 根据不同标准对股价排序
Stock Quote (股价)	无 (目前还没有)	公司名 股票代码 当前价格等等	分析 买入 卖出

3.1.3 身处对象世界中

程序员从过程性思维转而采用面向对象思路时，通常会有所顿悟，发现需要将属性和行为结合放入对象中。有些程序员会再次查看他们以前所做的程序设计，并将某些部分重新编写为对象。另外一些程序员可能会把原来的所有代码全盘丢掉，重新启动项目来建立一个完全面向对象的应用。

要基于对象来开发软件，对此主要有两种方法。对某些人来说，对象只是对数据和功能的一个很好的封装。这些程序员会在他们的程序中多处使用对象，以使代码更加可读，更易于维护。采用这种方法的程序员将独立的代码块取出，而代之以对象，就像是外科医生植入一个起搏器一样。就其本身而言，这种方法并没有不当的地方。这些人把对象视作一种工具，在许多情况下这个工具都能带来好处。一个程序中的某些部分确实“感觉像是一个对象”，股价便是如此。这些部分可以独立出来，而且可以按真实世界中的说法来描述。

另外一些程序员则会完全采纳 OOP 范式，把所有一切都转变成对象。在他们看来，有些对象对应于真实世界中的事物，比如一个桔子或一个股价，而另外一些对象则封装了更为抽象的概念，如分拣排序器或者完成撤销工作的 undo 对象。理想的方法可能介于这两个极端之间。你的第一个面向对象程序很可能只是一个传统的过程性程序，其中散布了一些对象。你也可能会整个地从头开始，把所有东西都建立为对象，从表示 int 的类到表示主应用的类，一切都纳入到类（对象）中。不过，经过一段时间之后，可能会得到一个合适的折衷方案。

过度对象化

不要让你的开发小组成员把每一个细微琐碎的地方都转成对象，他们会为此苦不堪言，这与设计一个创造性的面向对象系统有着严格的界线。弗洛伊德曾经说过，有时变量只是一个变量而已。这句话的含义再清楚不过了。

也许你正在着手设计另一个可能畅销的连珠 (Tic-Tac-Toe) 游戏。对此，你打算完全遵循 OOP 方法，所以你坐下来，喝着咖啡，在一个笔记本上大致勾画出类和对象。在这样一个游戏中，通常有一个对象统管游戏全程，这个对象能够检测出谁是赢家。要表示游戏棋盘，可能会考虑采用一个 Grid 对象，由它记录每一步做的记号以及记号的位置。实际上，这个网格的组件可以是 Piece 对象，每个 Piece 对象表示一个 X 或一个 O。

别着急，先退一步看！这个设计打算专门有一个类来表示一个 X 或一个 O。这就可能存在着过度对象化。毕竟，拿一个 char 表示 X 或 O 不也很好吗？更好的做法是，为什么 Grid 不直接使用一个二维的枚举类型数组呢？Piece 对象是不是只会让代码更复杂呢？表 3-2 表示了原先提出的棋子类。

表 3-2

类	相关组件	属 性	行 为
Piece	无	X 或 O	无

这个表没有多少内容，这就充分体现出，将棋子设计为一个完整的对象可能粒度过细了。

另一方面，提前做打算的程序员可能会争辩说，Piece 作为一个类尽管现在看来有些太“瘦”，但把它做成对象有利于以后的扩展，而且这也没有什么真正的损失。也许以后这要成为一个图形应用，而 Piece 类若能支持绘制行为可能很有用。还可以增加一些属性，如 Piece 的颜色，Piece 最近是否移动过等等。

显然，对此并没有绝对正确的答案。重点在于，当你设计应用时应当考虑到这些问题。要记住，对象的存在是为了帮助程序员管理他们的代码。如果使用对象只是为了使代码显得“更面向对象”，就肯定存在问题了。

### 过于一般的对象

前面指出的是将本来不应当是对象的东西建立为对象，与此相比，也许更糟糕的是存在过于一般的对象。所有学习 OOP 的学生都会从“桔子”之类的例子开始入手，这些东西确实是对象，这一点毋庸置疑。在实际的编码中，对象则可能相当抽象。许多 OOP 程序都有一个“应用对象”，不过应用并不是你在实际世界中看得到摸得着的东西。但是，把应用表示为一个对象可能很有用，因为应用本身有一些特定的属性和行为。

所谓过于一般的对象是指，对象根本没有表示任何特定的东西。程序员可能想建立一个灵活的或者可重用的对象，但是最后得到的可能只是一个莫名其妙的对象。例如，假设有一个组织和显示媒体的程序，它可以建立照片目录、对电子音乐集进行组织，还可以用作个人杂志。如果采用过于一般的方法，会把所有这些东西都认为是“媒体”对象，并建立一个类来满足所有媒体格式。它可能有一个名为“data”（数据）的属性，其中包含图像、歌曲或杂志某项内容的原始二进制数据，具体是什么取决于媒体的类型。它还可能有一个名为“perform”（完成）的行为，这个行为会适当地绘制图像、播放歌曲，或者打开杂志的某项内容以供编辑。

从这个类的属性和行为的字面上就能暗示出它可能太过一般了。“数据”一词本身没有太多的实际含义，之所以必须使用这样一个一般性的词，原因在于这个类做了过分扩展，要用于三个完全不同的用途。类似地，“完成”行为也会在三种不同的情况下做出截然不同的事情。最后要说明的是，这个设计为什么会过于一般，这是因为“媒体”并不是一个特定的对象。用户界面中没有这个对象，在实际生活中也找不到这个对象，甚至在程序员看来它也不是一个对象。要看一个类是不是太过一般了，有一个主要的线索，即是否将程序员脑海中的多种思想全都纠集在一起而成为一个对象，如图 3-2 所示。

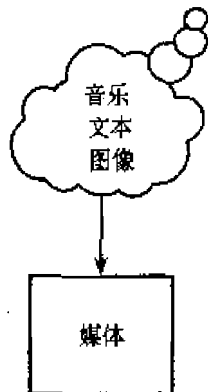


图 3-2

### 3.1.4 对象关系

作为一个程序员，肯定会遇到这样一些情况：多个类有一些共同的特性，或者至少看上去这些类相互之间存在某种关联。例如，尽管在一个数字目录程序中创建一个“媒体”对象来表示图像、音乐和文本的做法太过一般，但是这些对象确实有一些共同的特性。你可能希望这些对象都能记录最后一次修改时的日期和时间，或者希望它们都支持一种删除行为。

面向对象语言为处理对象之间的这些关系提供了大量机制。难点在于要理解关系到底是什么。主要有两种对象关系，一种是 *has-a* 关系，还有一种是 *is-a* 关系。

#### has-a 关系

参与一个 has-a 或聚集（aggregation）关系的对象都遵循以下模式，即 A 有一个 B，或者 A 包含一个 B。在这种关系中，可以把一个对象看作是另一个对象的一部分。如前面所定义的，组件一般就表示一种

has-a 关系，因为组件描述了构成其他对象的对象。

在实际中，动物园和一只猴子之间的关系就是这种 has-a 关系。可以说动物园有一只猴子，或者动物园中包含一个猴子。如果要用代码模拟动物园，可能有动物园对象，它会有一个猴子组件。

通常，考虑用户界面的情况将有助于理解对象关系。这是因为，尽管并非所有 UI（用户界面）都采用 OOP 实现（不过目前来讲，大多数 UI 都是如此），但是屏幕上的可视化元素都非常适于实现为对象。在 UI 领域中，可以打个比方，窗口包含一个按钮，这就是一个 has-a 关系。按钮和窗口完全是两个单独的对象，但是它们显然存在某种形式的关联。由于按钮在窗口内部，我们可以说窗口有一个按钮。

图 3-3 显示了真实世界和用户界面中的一些 has-a 关系。

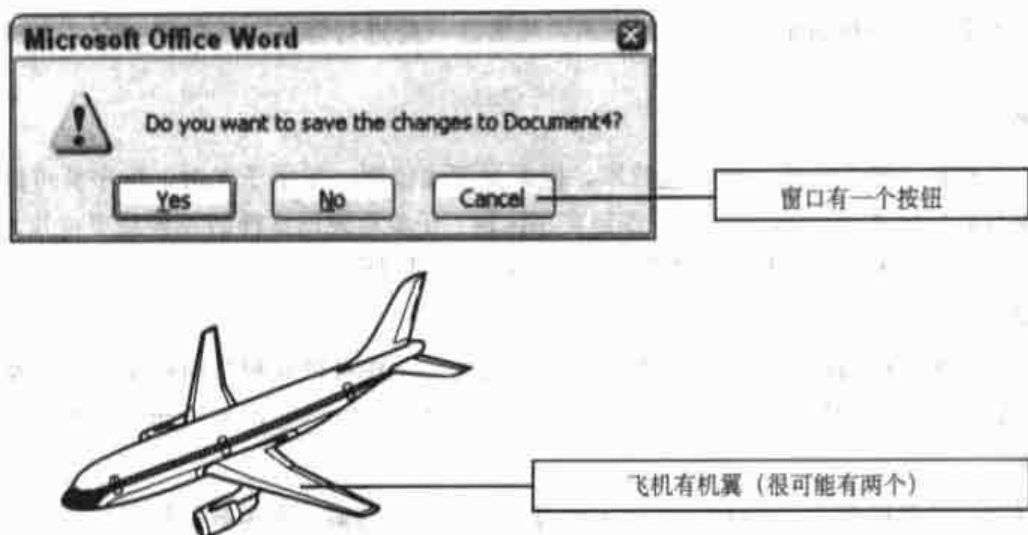


图 3-3

### is-a 关系（继承）

is-a 关系是面向对象程序设计中的一个相当基本的概念，它有很多名字，包括派生（subclassing）、扩展（extending）和继承（inheriting）。真实世界的对象有属性和行为，类正是对这一事实进行建模。这些对象可能以层次体系加以组织，继承则是对这一事实进行建模。这些层次体系就指示出了 is-a 关系。

基本说来，继承遵循以下模式：A 是一个 B，或者 A 实际上很像 B，这一点可能很复杂。还是考虑简单的情况，再来看前面的动物园，不过假设动物园里除了猴子之外还有一些其他的动物。单凭这一句话就已经建立了一个关系，猴子是一个动物。类似地，长颈鹿是一个动物，袋鼠是一个动物，企鹅是一个动物。如果你意识到，猴子、长颈鹿、袋鼠和企鹅都有某些共同的东西，就能发现继承的神奇之处了。这些共同性正是动物的一般特性。

对于程序员来说，继承的含义是指，可以定义一个 Animal 类，其中封装每种动物都有的属性（体态大小、生活地区、食物等等）和行为（移动、吃、睡觉）。特定的动物（如猴子）则作为 Animal 的子类，因为猴子包含动物的所有特性（要记住，猴子就是一个动物再加上另外一些突出的特性，正是这些特性使猴子与其他动物有所不同）。图 3-4 显示了动物的一个继承图。箭头指示了 is-a 关系的方向。

猴子和长颈鹿是不同类型的动物，与此类似，用户界面通常也有类型不同的按钮。例如，复选框就是一种按钮。假设按钮是一个可以单击、可以完成某个动作的 UI 元素，Checkbox 类则扩展了 Button 类，并增加了状态，即复选框是否被选中。

将类加入 is-a 关系时，这样做的一个目标是将共同的功能置于超类（superclass）中，所谓超类就是其他类所扩展的类。如果发现所有子类都有一些很相似的代码，或者几乎完全相同的代码，就可以考虑



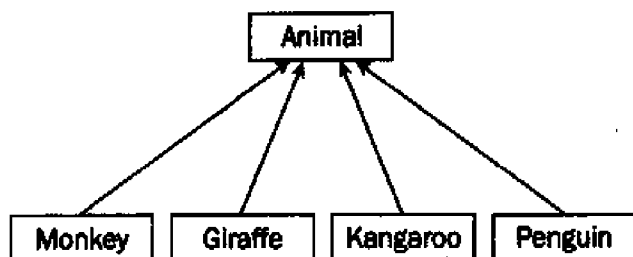


图 3-4

将这样的部分或全部代码移到超类中。如此一来，只需在一处进行修改，以后子类会“自然”地得到共享的功能。

#### 继承技术

前面的例子提到了继承中使用的一些技术，但未做正式说明。派生子类时，程序员可以采用多种方法使一个对象与其父对象（parent object）或超类相区别。子类会使用这样的一种或多种技术，而且通过“A 是一个怎样的 B”（A is a B that）之类的句子，就可以识别出子类来。

#### 增加功能

子类可以增加额外的功能来扩充其父类。例如，猴子是一种可以在树之间摇来晃去的动物。除了拥有 Animal 的所有行为外，Monkey 类还有一个“在树之间摇摆”（swing from trees）的行为。

#### 替换功能

子类可以完全替换或覆盖（override）其父类的一个行为。例如，大多数动物都是走着移动，所以你可能为 Animal 类提供了一个模拟走路的 move 行为。如果是这样，要知道，袋鼠就是跳着移动（而非走着行进）的动物。Animal 超类的所有其他属性和行为对袋鼠仍然适用，Kangaroo 子类只是会改变 move 行为的完成方式。当然，如果你发现需要把超类中的所有功能都替换掉，这就可能说明，在这种情况下派生子类根本不是一种合适的做法。

#### 增加属性

子类除了拥有从超类继承得到的属性外，还可以增加新的属性。企鹅不仅有动物的所有属性，还有一个喙大小（beak size）的属性。

#### 替换属性

C++ 提供了一种覆盖属性的方法，这与覆盖行为的方法很类似。不过，一般不应当这样做。重要的是不要把下面两个概念搞混了，即替换一个属性是一回事，而子类有不同的属性值又是另一回事。例如，所有动物都有一个 diet 属性，这说明它们都要吃东西。猴子爱吃香蕉（bananas），而企鹅要吃鱼（fish），不过无论是猴子还是企鹅都没有替换 diet 属性，只是为该属性所赋的值存在差别。

#### 多态与代码重用

多态（Polymorphism）概念是指，可以交替地使用遵循一组标准属性和行为的对象（译者注：这句话可以理解为：程序中会交替出现不同的对象，而且这些对象对于标准属性和行为可以有不同的实现）。类定义相当于对象和与之交互的代码（即使用对象的代码）之间的一个合约。根据定义，任何 Monkey 对象都必须支持 Monkey 类的属性和行为。这个概念还可以延伸到超类。由于所有猴子都是动物，所有 Monkey 对象还要支持 Animal 类的属性和行为。

多态是面向对象程序设计中的一大妙处，因为它充分利用了继承的精神。在对动物园的模拟中，我们可以编写程序循环处理动物园中的所有动物，让每个动物移动一次。由于所有动物都是 Animal 类的一

员，它们都知道如何移动。有些动物覆盖了移动行为，这也是设计中最“酷”的地方，我们的代码只是告诉每个动物要移动，而无需知道（也不必关心）这是一种什么动物。每个动物都会按照它所知道的方式去移动。

除了多态外，派生子类还有另一个理由。通常，这样做只是为了充分利用既有的代码。例如，如果需要一个类来播放带回音效果的音乐，而你的同事已经写过一个类，可以不带任何效果地播放音乐，你就能扩展现有的类，并加入新的功能。is-a 关系仍然适用（回音特效音乐播放器也是一个音乐播放器，只是增加了回音效果而已），不过，你可能不希望交替地使用这些类。你最后要得到的是两个单独的类，并用于程序中完全不同的部分（或者可能甚至在完全不同的程序中使用），它们之所以存在关联，只是想避免重复做同样的工作。

#### has-a 与 is-a 间的清晰界线

在真实世界中，很容易区别对象之间的 has-a 和 is-a 关系。没有人会说桔子有一个水果，正常的说法是：桔子是一种水果（is a）。在代码中，有时事情则并非这么一目了然。

下面考虑一个表示散列表的假想的类。散列表是一种数据结构，可以高效地建立键与值的映射。例如，一家保险公司可能使用一个 Hashtable 类将成员 ID 映射至成员名，这样给定一个 ID 时，就能很容易地找到相应的成员名。成员 ID 是键（key），而成员名就是值（value）。

在一个标准的散列表实现中，每个键都有一个值。如果 ID 14534 映射至成员名“Kleper, Scott”，它就不能将这个 ID 又映射至成员名“Kleper, Marni”。在大多数实现中，如果想要为一个已经有值的键增加另一个值，前一个值就会丢掉。换句话说，如果 ID 14534 映射至“Kleper, Scott”，而你又想把 ID 14534 分配给“Kleper, Marni”，那么 Scott 的保险就没有了，下面显示了对假想散列表 enter() 行为的两个调用，以及这两个调用之后所得到的散列表内容。hash.enter 有些类似于 C++ 的对象语法。可以认为这表示“使用 hash 对象的 enter 行为。”

```
hash.enter (14534, "Kleper, Scott");
```

键	值
14534	"Kleper, Scott" [字符串]

```
hash.enter (14534, "Kleper, Marni");
```

键	值
14534	"Kleper, Marni" [字符串]

假设有一个类似于散列表的数据结构，不过对应一个给定键允许有多个值，不难想像出这样一个数据结构的使用。在保险例子中，如果一个家庭对应一个 ID，这样就可能有多个名字（一个家庭中的多个成员）对应同一个 ID。由于这种数据结构与散列表很相似，最好能以某种方式利用散列表的功能。散列表对应一个键只能有一个值，但是这个值可以是任何东西。值可以不是一个字符串，而是一个集合（如数组或列表），其中包含对应键的多个值。每次为一个既有的 ID 增加一个新成员时，只需把新的成员名增加到集合中。由以下调用序列可以了解这种做法。

```
Collection collection;           // Make a new collection.
collection.insert("Kleper, Scott"); // Add a new element to the collection.
hash.enter(14534, collection);    // Enter the collection into the table.
```

键	值
14534	{"Kleper, Scott"} [集合]

```
Collection collection = hash.get(14534); // Retrieve the existing collection.
collection.insert('Kleper, Marni');      // Add a new element to the collection.
hash.enter(14534, collection);           // Replace the collection with the updated one.
```

键	值
14534	{"Kleper, Scott", "Kleper, Marni"} [集合]

如果使用集合而不是字符串，这会很繁琐，而且需要大量重复性的代码。更好的做法是把这种多值功能包装在一个单独的类中，可以称之为 MultiHash。MultiHash 类与 Hashtable 的工作类似，不过其底层实现中，将每个值存储为一个字符串集合，而不是单一的一个串。显然，MultiHash 与 Hashtable 存在某种关联，因为它还是在使用一个散列表存储数据。所不明确的是，这要构成一种 is-a 关系还是一种 has-a 关系。

先来考虑 is-a 关系，假设 MultiHash 是 Hashtable 的子类。它必须对“向表中增加一项”这个行为进行覆盖，从而可以创建一个集合并增加新元素，或者是获取既有的集合并增加新元素。此外还要覆盖“获取值”的行为。例如，可以把对应一个给定键的所有值追加到一个串中。看上去这是一个很合理的设计。尽管它覆盖了超类的所有行为，但是由于子类中使用了原来的行为，因此也算利用了超类的行为。这种方法如图 3-5 所示。

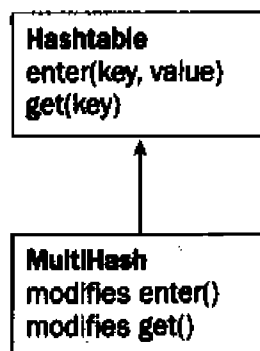


图 3-5

下面再把它考虑为一个 has-a 关系。MultiHash 不再作为子类，它是一个单独的类，不过其中包含 (contain) 一个 Hashtable 对象。它可能有一个与 Hashtable 很类似的接口，不过无需完全相同。在底层实现中，用户向 MultiHash 增加内容时，所增加的内容实际上会包装在一个集合中，并放在 Hashtable 对象里。看上去似乎也很合理，如图 3-6 所示。

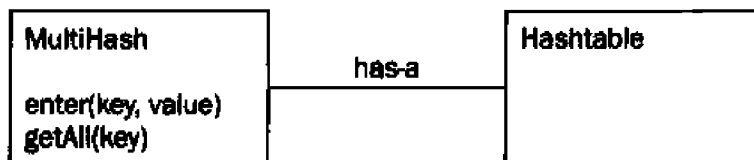


图 3-6

那么，哪一种解决方案才算正确呢？对此没有明确的答案，不过本书作者之一曾经写过一个 MultiHash 类（用于成品阶段），他就把这看作是一种 has-a 关系。其主要原因是，这样可以对所提供的接口进行修改，而不必操心维护散列表功能。例如，在图 3-6 中，get 行为修改为 getAll，以表明它要得到 MultiHash 中对应某个键的所有值（但不必修改 Hashtable）。另外，基于 has-a 关系，无需操心散列表带进来的所有功能（即不必了解这些功能的细节）。例如，如果散列表类支持一种行为，可以得到总共有多少个值，利用这一点，MultiHash 就能直接报告集合的个数，除非 MultiHash 有意要覆盖。

不过，有人可能会提出很充分的理由，认为 MultiHash 实际上就是带有一些新功能的 Hashtable，而

这应当是一个 is-a 关系。实际上这里的关键在于，在这两种关系之间，有时存在着一个清晰的界线，应当考虑类将如何使用，还要看看你所构建的类是否充分利用了另一个类的某些功能，或者实际上它就是另一个类，只不过增加或修改了某些功能而已。

针对 MultiHash 类可以采用的两种方法，表 3-3 列出了支持和反对这两种方法的理由。

表 3-3

	is-a	has-a
支持的理由	<ul style="list-style-type: none"> <li>• 基本说来，MultiHash 与 Hashtable 相比，是存在不同特性的同一个抽象</li> <li>• 它提供了与 Hashtable（几乎）相同的行为</li> </ul>	<ul style="list-style-type: none"> <li>• MultiHash 可以有任何有用的行为，而无需操心散列表有什么行为</li> <li>• MultiHash 的实现可以修改为使用其他结构（不使用 Hashtable），而不会改变外部行为</li> </ul>
反对的理由	<ul style="list-style-type: none"> <li>• 根据定义，散列表对应每个键只有一个值。要说 MultiHash 是一个散列表，显然与上述定义相悖！MultiHash 完全覆盖了 Hashtable 的两个行为，从这一点上看，也强有力地表明这种设计存在问题</li> <li>• Hashtable 的一些未知或不合适的属性或行为可能会“带入到”MultiHash 中</li> </ul>	<ul style="list-style-type: none"> <li>• 从某种意义上说，MultiHash 提供了新的行为，这也算是做了重复的工作</li> <li>• Hashtable 另外的一些属性和行为可能很有用</li> </ul>

#### not-a 关系

在考虑类存在哪一种类型的关系时，还要考虑这些类是否真的有关系。不要因为你面向对象设计的狂热，而带来大量完全不必要的类/子类关系。

某些东西在真实世界中显然是相关的，但在编写代码时并不存在真正的关系，此时就会出现一个陷阱。在现实中，Mustang 是一种 Ford 汽车，但仅凭这一点并不表示，在编写一个汽车模拟程序时，Mustang 一定要作为 Ford 的一个子类。OO 层次体系需要对功能关系建模，而不是对人为的关系建模。图 3-7 显示的关系作为本体或者作为层次来讲是有意义的，但是在代码中可能并不表示有意义的关系。

要想避免没有必要的派生子类，最好的方法是先粗略地完成设计。对于每个类和子类，写出你打算在其中放置（实现）哪些属性和行为。如果发现一个类没有自己的特定属性或行为，或者一个类的所有属性和行为都会被其子类完全覆盖，就应该重新考虑设计了。

#### 层次体系

就像类 A 可能是 B 的一个超类一样，B 也可能是 C 的一个超类。面向对象层次体系可以为诸如此类的多层关系建模。如果动物园中有更多的动物，模拟这样一个动物园时，可以把每种动物设计为一个公共 Animal 类的子类，如图 3-8 所示。

在编写上述各个子类时，可能会发现它们存在很多类似之处。如果出现这种情况，应当考虑将共同的地方放到一个公共的父类中。可以了解到，狮子（Lion）和豹子（Panther）有相同的移动方式，而且它们吃的东西也相同，这就表明可以为它们建立一个父类 BigCat。你还可以把 Animal 类进一步划分为包括水生动物（WaterAnimal）和有袋动物（Marsupial）。图 3-9 所示的层次设计更清楚地反映出这种共同性。

生物学家看到这个层次图时可能会不太满意，要知道，企鹅和海豚实际上不属于同一科。不过，由此可以很好地反映出，在编写代码时，对于真实世界中的关系和共享的功能关系，需要做适当的平衡。尽管两个东西在真实世界中没有太大的关联，但是在代码中它们之间可能存在一种 not-a 关系，因为它们没有共享功能。也可以简单地把动物划分为哺乳动物和鱼类，但是这样一来，怎么在它们的超类中建

立其共同性呢？

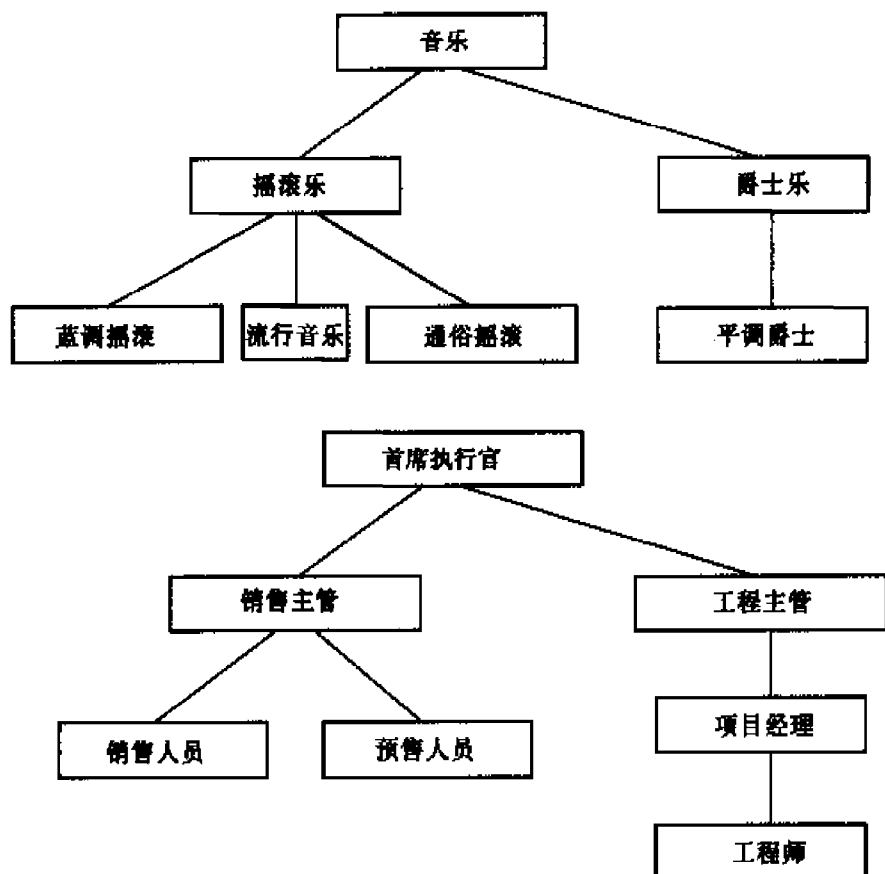


图 3-7

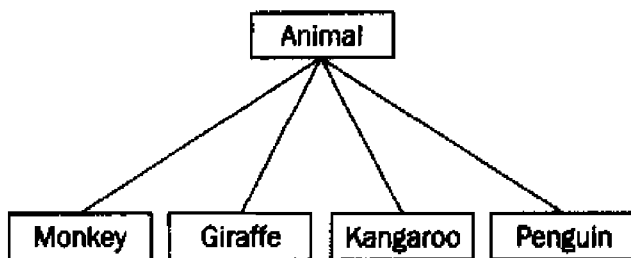


图 3-8

还有一点很重要，对于层次体系的组织可能还有其他的方法。前面的设计主要是按动物如何移动来组织。如果按动物的食物或高度来组织，所得到的层次体系就会又是一个样子。最后需要指出，如何使用类才是关键。这些需求可以指示出对象层次体系的设计。

好的面向对象层次体系要满足以下要求：

- 将类按有意义的功能关系加以组织。
- 将共同的功能放到超类中，从而支持代码重用。
- 避免子类过多地覆盖父类的功能。

#### 多重继承

到目前为止，每个例子都只有一条继承链。换句话说，一个给定的类至多有一个直接父类。并不一

定必须如此。通过多重继承，类可以有多个超类（译者注：更确切的说，多重继承是指类可以有多个直接父类）。

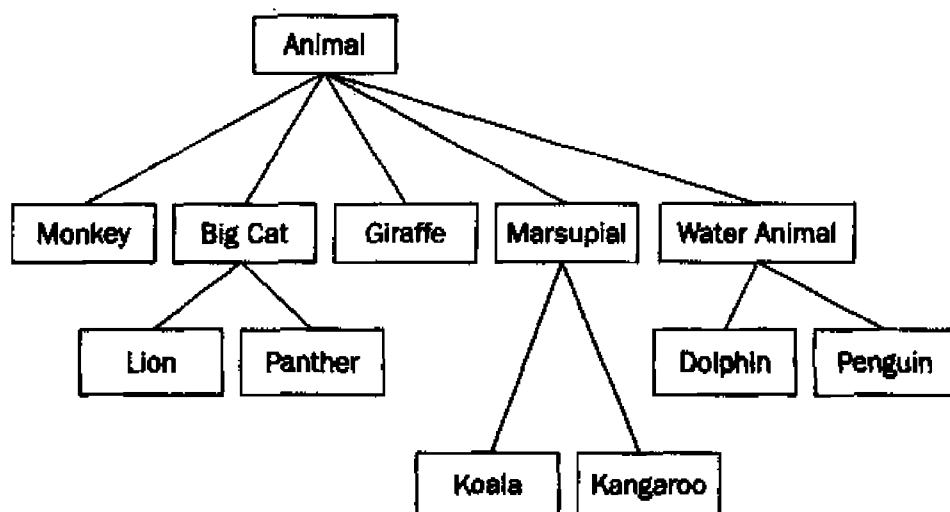


图 3-9

如果你认为动物在太多方面存在着差异，那么任何一个动物对象层次体系似乎都不太好，此时你可能就需要多重继承。利用多重继承，可以创建三个单独的层次体系，一个按大小组织，一个按食物组织，还有一个按移动方式组织。这样，每个动物都能选择其中的一个体系结构。

图 3-10 显示了一个多重继承设计。这里仍然有一个名为 Animal 的超类，它进一步按大小进行了划分。另外根据动物的食物建立了一个单独的层次体系，第三个层次体系则按动物的移动方式组织。这样，每一类动物都是这三个类（Mover、Eater 和 Animal）的子类，对此用不同颜色的线显示。

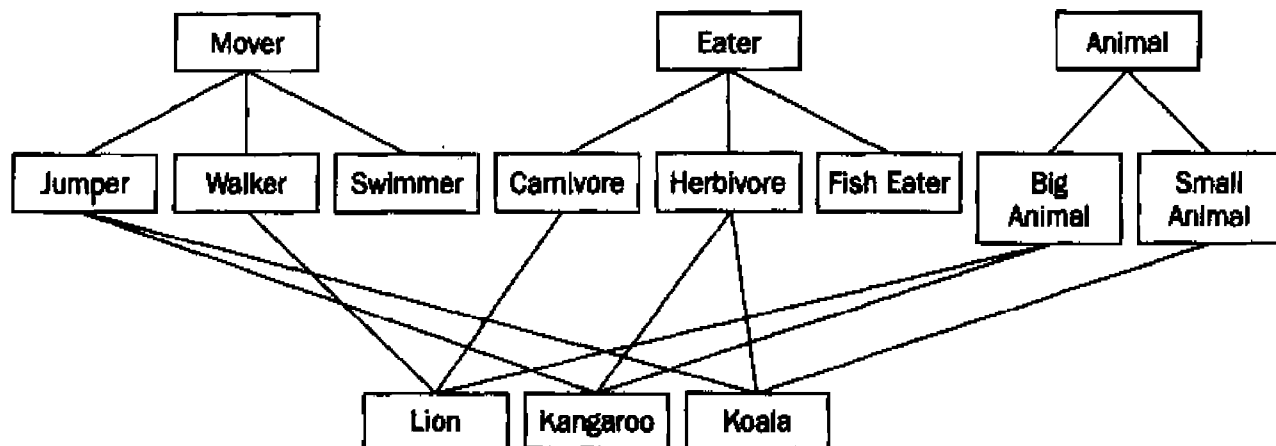


图 3-10

假设在用户界面中有一个图像，用户可以单击这个图像。这个对象看上去既是一个按钮，又是一个图像，因此其实现可以既派生 Image 类，又派生 Button 类，如图 3-11 所示。



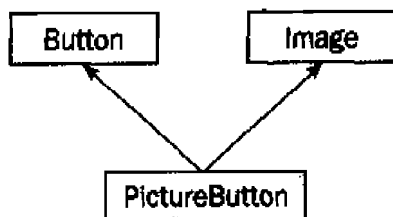


图 3-11

### 多重继承有什么坏处

许多程序员都不喜欢多重继承。C++ 对这种关系提供了显式支持，但是 Java 语言去掉了这个特性。对多重继承存在的批评，原因是多方面的。

首先，多重继承的可视化表示很复杂。如图 3-10 所示，即使是这样一个简单的类图，由于存在多个层次体系和交错的线，它也会变得非常复杂。建立类层次就是为了使程序员更容易理解代码之间的关系。基于多重继承，一个类可以有多个父类，这些父类之间并无关联。由于你的对象中有这么多类的“贡献”，你真的能搞清楚到底发生了什么吗？在真实世界中，我们往往不会认为对象有多个 is-a 关系。

其次，本来应该很清晰的层次结构可能会因为多重继承而变得混乱不堪。在动物例子中，如果转而采用一种多重继承方法，这说明 Animal 超类已经没多少意义了，因为描述动物的代码现在已经分别放到了三个单独的层次体系中。尽管图 3-10 中所示的设计显示了三个清晰的层次体系，但不难想像它们将成为怎样的一团乱麻。例如，如果发现所有 Jumper 不光都跳着走，而且吃的东西也一样，会怎么样呢？因为这三个层次体系都是独立的，我们只能再增加另一个子类，除此以外，再没有别的办法可以结合移动方式和食物这两个概念了。

第三，多重继承的实现很复杂。如果两个超类以不同方式实现了同一个行为会怎么样呢？能不能让这两个超类本身又作为另一个共同超类的子类呢？由于存在这样一些可能性，就会导致实现很复杂，因为在代码中建立这么错综复杂的关系不光对编写代码的人来说是个难题，对于阅读代码的人来说也同样很费劲。

其他一些语言可能去除了多重继承，其原因是：通常多重继承是可以避免的。通过重新考虑层次体系，或者使用第 26 章介绍的某些设计模式，不必引入多重继承就能准确地把握项目的设计。

### 混合类

混合 (mix-in) 类表示了类之间的另外一种关系。在 C++ 中，混合类的实现语法与多重继承很像，不过其语义则截然不同！混合类回答了这样一个问题“这个类还能做什么？”，而且答案中往往是“能够……”。基于混合类，可以为一个类增加功能，而不必建立一个完整的 is-a 关系。

再来看动物园的例子，你可能想表达这么一种概念，有些动物“可以逗着玩”。也就是说，来动物园的游客可以和这样一些动物嬉戏，却不会被咬伤或打伤。你可能希望所有可以逗着玩的动物都支持一个行为“逗着玩”。由于可以逗着玩的动物并没有其他共同之处，所以你不愿打破原先设计的层次体系，这么看来，Pettable 就完全可以作为一个混合类。

混合类通常用在用户界面中。我们一般不会说一个 PictureButton 类同时是一个 Image 和一个 Button，而会说这是一个可以单击的 (Clickable) Image。桌面上的一个文件夹则是一个可拖放 (Draggable) 的 Image。在软件开发中，我们造出了许多这种有意思的形容词（译者注：这是指 Clickable、Draggable 等等表示“可以做什么”的词汇，在英语中本来并没有这样的单词）。

混合类和超类之间存在着区别，这种区别并不是代码上的差别，更反映在你将如何考虑类。一般来说，混合类比多重继承更易于理解，因为混合类往往限制在某个范围内。Pettable 混合类只是为所有既有

的类增加一个行为而已。Clickable 混合类可能只是增加了“鼠标按下”和“鼠标放开”行为。另外，混合类一般没有太大的层次体系，所以不存在功能的相互交错。

### 3.1.5 抽象

在第2章中，你已经了解了抽象的概念，这是指将具体实现与对实现的访问方法相分离。抽象是一个很好的思想，前面已经分析过为什么抽象很有意义。这也是面向对象设计的一个基本部分。

#### 接口和实现

抽象的关键就是将接口 (interface) 与实现 (implementation) 有效地分离。所谓实现，是指为完成所需完成的任务而编写的代码。接口则是指其他人用什么方法来使用你的代码。在C中，如果头文件描述了你所写的库中的函数，这个头文件就是一个接口。在面向对象程序设计中，类的接口是一个由可公共访问的属性和行为所组成的集合。

#### 确定对外的接口

在设计一个类时，会出现这样一个问题，即其他程序员如何与你的对象交互。在C++中，类的各个属性和行为可以限定为 public、protected 或 private。public 是指其他代码可以访问这个属性或行为。protected 表示其他代码不能访问此属性或行为（译者注：但子类可以访问父类的 protected 属性或行为）。private 则是一个更严格的控制，它表示不仅一般的其他代码不能访问这些属性或行为，另外，即使是子类也不允许访问。

设计对外的接口实际上就是选择哪些属性和行为要置为 public。与其他程序员共同完成一个大型项目时，应当把设计对外接口看作是开发中的一个重要过程。

#### 考虑受众

设计对外接口的第一步就是考虑你是为谁设计这个接口。使用这个接口的人（受众）是开发小组中的一个成员吗？你是不是只打算自己使用这个接口？这个接口会不会被你公司之外的某个程序员使用呢？会是一个顾客，还是一个下级承包人？这一步除了可以确定谁将利用这个接口，还会明确你的一些设计目标。

如果只是你自己使用这个接口，那么设计时就可能有更大的自由度。你在利用这个接口时，可以对其进行修改来满足自己的需求。不过，一定要谨记工程小组中的角色可能会发生变动，而且总有一天很可能其他人也会使用这个接口。

为其他内部程序员（即小组中的其他成员）设计接口则稍有不同。从某种意义上说，接口也就是你与这些程序员之间的一个合约。例如，如果你实现了程序的数据存储库组件，其他人要依赖于这个接口来支持某种操作。就需要明确小组中的其他人要使用你的类做些什么。他们需要版本控制吗？他们能存储哪些类型的数据？作为一个合约，你应当把接口看作是一种不太灵活的东西（译者注：即不太会改变）。如果在编写代码之前协商好了接口，而在代码写完之后你又决定修改接口，那么其他程序员肯定会对你提出抱怨。

如果客户是一个外部顾客，你就要基于另外一组迥然不同的需求开始设计。理想情况下，可能会要求目标顾客指定你的接口要提供哪些功能。你不仅要考虑他们希望得到的特定功能，还要考虑他们将来可能希望得到哪些功能。接口中的用词应当与顾客熟悉的词相对应，而且在写文档时必须从看文档人的角度来考虑。在设计中，不应当出现玩笑话、代码名和程序员的行话。

#### 考虑用途

编写接口的原因有很多。在实际编写代码前，甚至在确定要提供什么功能之前，需要理解接口的用途。

### 应用编程接口 (API)

应用编程接口 (Application Programming Interface, API) 是一种外部可见的机制, 利用 API 可以在另一个上下文中扩展一个产品, 或者使用该产品的功能。如果说内部接口是一个合约, 那么 API 就相当于铁定的法律。一旦有你公司之外的人使用你的 API, 他们肯定不希望 API 出现改动, 除非你增加了新的功能能够对其提供帮助。由此说来, 在让顾客使用你的 API 之前, 一定要特别小心地规划 API, 并与顾客讨论交流。

在设计 API 时, 主要是要权衡 API 的易用性和效率。由于使用接口的目标用户并不熟悉产品的内部工作原理, 如何使用 API 的学习曲线应当是逐渐上升的 (即不应过陡)。毕竟, 你的公司之所以向顾客提供这个 API, 就是为了让顾客能够使用它。如果太难于使用, 这个 API 就要算是一个失败。通常灵活性会与此背道而驰。你的产品可能有许多不同的用途, 而且你希望顾客能够充分利用你所提供的所有功能。不过, 你的产品也许能做很多工作, 如果能让顾客都能用到的话, 这样的 API 就会太过复杂了。

正如一则编程箴言所说的: “好的 API 可以让容易依然容易, 让困难的事情也变为可能。” 也就是说, API 应该有一个简单的学习曲线。通过 API, 大多数程序员想做的工作应该都能够做到。不过, API 还支持更高级的使用, 尽管一般情况下都要求 API 具有简单性, 但是在极少情况下, 出于某种折衷考虑, 复杂的 API 也是可以接受的。

### 工具类或库

通常, 你的任务就是开发某个特定的功能, 以便在应用中别的地方加以使用。这可能是一个随机数据库或者是一个日志记录类。在这些情况下, 接口是比较容易确定的, 因为你要提供大多数或者所有的功能, 而最好不暴露有关实现的太多内容。通用性是一个需要考虑的重要问题。由于这个类或库是通用的, 必须在设计时就考虑到一组可能的用例。

### 子系统接口

你可能要设计应用中两个主要子系统之间的接口, 如访问数据库的机制。在这些情况下, 将接口与实现相分离是至关重要的, 这是因为, 在实现完成之前, 其他程序员很可能已经基于你的接口开始完成他们的实现了。在开发子系统时, 首先要考虑这个子系统的一个主要用途是什么。一旦明确了子系统所要完成的主要任务, 再来考虑它的特殊用途, 以及应当如何提供给代码中的其他部分。要从总体角度考虑, 而不要过分深陷于实现细节当中。

### 组件接口

你定义的大多数接口都可能比子系统接口或 API 要小。这些对象会在你所写的其他代码中使用。在这些情况下, 主要存在这样一个陷阱, 接口可能会不断扩大, 以至于最后发展到无法控制的地步。尽管这些接口只是你自己使用, 但是不要这样考虑这些接口, 而要认为这些接口还可能为别人所用。与子系统接口一样, 需要考虑每个类的一个主要用途, 对于与这个用途无关的功能, 对外提供这样一些功能时要务必谨慎。

### 为将来做考虑

在设计接口时, 要考虑到将来的情况。是不是今后几年你都要采用这个设计? 倘若如此, 就可能需要提供一个插件架构来留出扩展的空间。你是否发现别人可能将你的接口另作他用 (即并非原先所设计的用途)? 如果是这样, 就要与他们交流, 对他们的用例有更充分的认识。还有一种做法是以后再重新编写接口, 或者更糟糕的, 发现需要新功能时就随时加上, 根本没有任何计划, 这样最后就会得到一个杂乱无章的接口。不过千万注意, 如果过分强调通用性, 则会落入另一个陷阱。如果不清楚将来会做何使用, 就不要设计一个“万能”的日志记录类。

### 设计一个成功的抽象

要设计好的抽象，经验和反复是关键。只有经过多年编写和使用抽象的历练，才能设计出真正的好接口。你遇到其他抽象的时候，一定要记住其中哪些可行，哪些不可行。你上周使用 Windows 文件系统 API 时发现其中缺少了什么？如果是自己写网络包装器，与你的同事编写的相比，你会做哪些不同处理？一般来说，最早在纸上设计的接口往往不是最好的接口，所以一定要反复调整。让相关的人了解你的设计，并得到他们的反馈。也许你已经开始编写代码了，但也不要因此害怕修改抽象，即使这可能意味着其他程序员要做相应调整。他们应该能意识到，从长远来看，好的设计对每个人都有利。

与其他程序员交流你的（新）设计时，有时需要稍做一些解释。开发小组的其他人可能并未发现前一个设计（即修改前的设计）有问题，也有可能认为采用你的新方法的话，他们要做太多的工作。在这种情况下，一方面要坚持你的工作，另一方面还要在适当的地方采纳他们的思想。如果他们还存在异议，可以提供一个清楚的文档和一个示例代码，这往往就能把他们“征服”。

要当心单类（single-class）抽象。如果你写的代码层次过深，就要考虑可以另外建立哪些辅助类来“辅佐”主接口。例如，如果你提供了一个接口来完成一些数据处理，还可以考虑编写一个结果对象，由它提供一种简便方法来查看和解释结果。

在可能的情况下，应当将属性转变为行为。换句话说，不要让外部代码直接操纵类中的底层数据。一些粗心或草率的程序员可能会把一个“兔子”对象的高度设置成负数，你肯定不希望这样。所以正确的做法应当是，建立一个“设置高度”的行为，其中要完成必要的越界检查。

在此还要重申反复的意义，因为这是最重要的一点。要搜寻对设计的反馈，并做出反应，如果必要还要进行修改，并从错误中学习。

第5章介绍了设计接口和可重用代码的更多原则。

## 3.2 小结

在本章中，你对面向对象程序的设计有了一定的认识，却没有过多地涉及具体的代码。你所学到的概念几乎可以应用到任何一种面向对象语言中。其中一些概念对你来说可能只是复习，而另外一些则可能对你以前熟悉的概念做了全新的诠释，需要全新的理解。通过本章的学习，你也许会采用某些新方法来解决原有的问题，也可能会引用新的证据来支持你一直以来在开发小组中倡导的一些概念。即使在代码中没有用过对象，或者用得很少，但对于如何设计面向对象程序，你现在所掌握的可能比许多有经验的 C++ 程序员还要多。

学习对象之间的关系非常重要，这是因为对象之间如果有清晰的关系，将有利于代码复用，并能减少混乱，还不仅如此，你会在一个团队中工作，这也是一个原因。如果对象以合理的方式关联，这样的对象将更易于阅读和维护。设计程序时，你可能会以 3.1.4 节作为参考。

最后，你还学习了如何创建成功的抽象，并了解了两个最重要的设计考虑——谁来使用此设计，以及设计的用途。第4章将展开介绍抽象的开发，包括诸如代码重用、思想重用等主题，以及可供使用的一些库。

## 第 4 章 基于库和模式的设计

有经验的 C++ 程序员不会从头开始开发一个项目，而是会从多种不同来源获取代码，如标准模板库、开源库、本单位的专用代码基，以及他们在以前项目中编写的代码。另外，好的 C++ 程序员会重用一些方法或策略来解决各种常见的设计问题。从用于以往项目的某种技术，到正式的设计模式，都属于这样一些策略。本章将介绍设计程序时如何将既有的代码和策略考虑在内。

第 2 章已经介绍了重用的原则，并解释过这种原则不仅适用于代码重用，还适用于思想重用。本章将详细介绍这个内容，在此会提供一些特定的细节和策略，你可以在程序设计中使用时使用这些策略。读完本章后，应该了解以下内容：

- 可供重用的不同类型的代码
- 代码重用的优点和缺点
- 重用代码的一般策略和原则
- 开源库
- C++ 标准库
- 设计技术和模式

### 4.1 重用代码

应当在设计中充分地重用代码。为了更准确地把握这个原则，需要理解可以重用哪些类型的代码，还要了解代码重用中存在哪些折衷考虑。

#### 4.1.1 有关术语

在分析代码重用的优点和缺点之前，先要明确有关的术语，并划分重用代码的类型，这会很有帮助。可供重用的代码主要有三大类：

- 以前你自己写的代码
- 你的同事编写的代码
- 你所在组织或公司之外的某个第三方编写的代码

还可以采用多种方式对你使用的代码加以组织：

- 独立的函数或类。重用你自己的代码或同事所写的代码时，所遇到的一般都是这一类代码。
- 库。库（library）是用于完成一个特定任务（如解析 XML）或者处理一个特定领域（如加密）的代码集。使用第三方代码时，这些代码往往都采用库的形式。你可能用过一些简单的库，如 C 或 C++ 中的数学库，因此对于库应该并不陌生。另外一些通常由库提供的功能包括线程和同步支持、网络 and 图形等。
- 框架。框架（framework）是你设计程序所基于的一个代码集。例如，MFC 基类（Microsoft Foundation Class）就为创建 Microsoft Windows 的图形用户界面应用提供了一个框架。框架通常

决定了程序的结构。第 25 章提供了有关框架的更多信息。

程序要使用 (use) 库，而要纳入 (fit into) 框架。库提供了一些特定的功能，而框架则是程序设计和结构的基础。

还有一个词经常出现，这就是应用编程接口 (Application Programming Interface, API)。API 是一个库或面向某特定用途的代码体的接口。例如，程序员通常会谈到 socket API，这就是指 socket 网络库的对外接口，而并非库本身。

尽管人们会交替地使用 API 和库这两个术语，但二者并不等同。库指的是实现，而 API 是指库的公开接口。

为简单起见，本章余下部分会用“库”一词来表示所有的重用代码，实际上这可以是一个库、框架，也可以是你同事提供的任何函数集。

#### 4.1.2 决定是否重用代码

从抽象的层次看，重用代码的规则很容易理解。不过，真正落到实处时，就有些含糊了。如何知道什么时候适于重用代码，要重用哪些代码？这往往存在一个折衷，究竟做何决定取决于具体的情况。不过，重用代码有一些一般性的优点和缺点。

##### 重用代码的优点

重用代码可以为你和你的项目提供很多好处。

- 首当其冲地，你可能无法编写 (或者不想编写) 要重用的代码。你真的想编写代码来处理格式化输入和输出吗？回答当然是否定的，正是出于这个原因，你才会使用 C++ 的标准 I/O 流。
- 重用代码可以节省时间。如果重用了代码，那么这些代码就无须自己编写了。另外，你的设计也将更简单，这是因为，应用中重用的那些组件无需你自行设计。
- 从理论上讲，与自己编写的代码相比，重用的代码需要的调试更少。自己写的代码很可能存在 bug，但一般可以认为库代码是没有 bug 的，因为库已经得到了充分的测试和使用。当然，这也存在例外，也有可能你选用的库编写得很糟糕，存在很多的 bug。
- 库可能还会处理更多的错误条件，而自己写代码时可能想不到这么多。在项目开始时，你可能会忘记处理一些生僻的错误或极端情况，等到以后出现问题时再去修正就会浪费很多时间。或者更糟糕的是，这些错误条件可能会作为 bug 暴露在你的用户面前。重用的库代码往往得到了充分的测试，而且已经由众多程序员使用过，因此可以认为库已经适当地处理了绝大多数错误。
- 与你为某个领域自行编写的代码相比，领域专家编写的重用代码更安全一些。例如，除非你是一个安全领域的专家，否则不要尝试编写自己的安全性代码。如果需要在程序中引入安全性或完成加密，可以直接使用一个库。假设自己编写安全性代码，如果处理不当，许多看似微小的细节则会破坏整个程序的安全性。
- 最后一点，库代码会持续地改进。如果重用代码，就会不断受益于这些改进，而无需自己动手！实际上，如果编写库的人适当地分离了接口和实现，只需升级库版本，不必修改与库的交互，就能获得库改进所带来的好处。真正好的升级只会修改底层实现，而不会修改接口。

##### 重用代码的缺点

遗憾的是，重用代码也存在一些缺点。

- 如果只使用自己写的代码，你就能很清楚地了解这些代码是如何工作的。如果使用了并非自己写



的库，就必须花些时间来理解接口和正确的用法，在此之后才能真正使用它。由于项目开始时花费这些额外的时间，设计和编写代码就会变慢（推迟）。

- 编写自己的代码时，它会做你想做的工作。但库代码可能并不能完全提供你需要的功能。例如，本书作者之一曾经就犯过一个错误，他没有注意到一个可扩展标记语言（eXtensible Markup Language，XML）解析库中存在一个显著的缺陷，就开始着手使用。乍一看这个库相当不错，它同时支持文档对象模型（Document Object Model，DOM）和面向 XML 的简单 API（Simple API for XML，SAX）解析模型，能够高效运行，而且没有许可费用。但直到真正编写代码时，才注意到这个库不支持按照文档类型定义（Document Type Definition，DTD）进行验证。
- 即使库代码确实提供了你所需的功能，但性能可能不能如你所愿。一般来讲库代码的性能都不好，而且针对你的特定用例，其性能可能相当差，或者根本未公开性能到底如何。另外，编写库或文档的人与你的观念可能有所不同，在判别性能好坏时他们可能有不同的标准。
- 使用库代码的话，在支持方面就像是打开了一个潘多拉盒子。如果发现库中存在一个 bug，该怎么办呢？你通常没有办法访问源代码，因此即使想修正错误也无从下手。如果你已经花了很多时间来学习库接口，以及如何使用这个库，很可能不舍得放弃它，但是你会发现，想在你的时间进度之内说服库开发人员修正这个 bug 可能很困难。另外，倘若你在使用一个第三方库，如果编写库的人不再对这个库提供支持，而你还没有放弃支持依赖于这个库的产品，又该如何是好呢？
- 除了支持问题，库还会带来许可问题。使用开源的库通常要求代码也是开源的。有时库还需要许可费用，在这种情况下，重新开发自己的代码可能更廉价一些。
- 对于重用代码还有一个考虑，这就是跨平台可移植性。大多数库和框架都是特定于具体平台的。例如，毫无疑问，MFC 框架主要用于 Microsoft Windows。即使是声称能跨平台的代码，在不同平台上也可能会表现出细微的差别。如果你想编写一个跨平台的应用，可能需要在不同的平台上使用不同的库。
- 开源软件也有自己的安全性问题。有些程序员出于安全性原因很忌讳使用开源代码。通过阅读程序的源代码，黑客（恶意攻击者）可能会发现并利用其中的 bug，如果不开放源码的话，这些 bug 可能不会被发现。
- 最后一点，重用代码需要一定的信任。不论是谁编写的代码，都必须相信他，认为他（或她）很好地完成了工作。有些人喜欢对项目的方方面面全盘控制，包括每一行源代码。本书作者之一就发现，很难完全相信不是自己写的库代码。不过，在软件开发中，这通常不是一个有益的观点。

### 综合来做出决定

既然已经熟悉了重用代码的术语、优点和缺点，应该已经有了更充分的准备，可以决定是否重用代码了。通常，这个决定是很显然的。例如，如果你想用 C++ 为 Microsoft Windows 编写一个图形用户界面（GUI），就应当使用诸如 MFC 的一个框架。你可能不知道如何编写底层代码在 Windows 中创建一个 GUI，而且更重要的是，你不想浪费时间来学如何做到这一点。在这种情况下，通过使用框架，就能省下很多人年（person-year）（译者注：人年是开发成本的一个度量单位）。

不过，在另外一些情况下，如何选择可能并不这么明确。例如，如果你对一个库或框架不太熟悉，而且只需要一个简单的数据结构，倘若花时间学习整个框架，又只是使用其中的一个组件（如果你自己编写这个组件，也不过花费几天的功夫），就很不值得了。

最后要说明的是，这个决定是一个主观选择，你要针对自己特定的需求来做出选择。最后往往会在两个时间之间有所权衡，即一方面是如果自行编写需要多少时间，另一方面是学习如何使用一个库来解决问题所需要的时间。请仔细考虑，针对你的特殊情况是否存在上述优缺点，并确定对你来说哪些因素

最重要。最后要记住，你的想法可能并非一成不变！

### 4.1.3 重用代码的策略

在使用库、框架、同事提供的代码或你自己的代码时，有一些需要谨记的原则。

#### 理解功能和局限性

花些时间来熟悉重用的代码。不仅要了解它的功能，还要了解它的局限性，这一点很重要。先从文档和公开接口或 API 开始。理想情况下，这对于理解如何使用代码已经足够了。不过，如果库并未将接口和实现清晰地分离，可能就要分析源代码本身了。另外，还可以与使用过这个代码而且能够做出解释的其他程序员交流。要先学习基本功能。如果这是一个库，它会提供哪些行为？如果这是一个框架，那么你的代码如何纳入这个框架？要派生哪些类？哪些代码需要自己编写？依据不同类型的代码，还要考虑一些特定的问题。

以下是要记住的对库或框架通用的几点：

- 此代码对多线程程序是否安全？
- 库或框架需要哪些初始化调用？它需要哪些清除工作？
- 这个库或框架依赖于另外的哪些库？

针对你使用的库调用，要记住以下几点：

- 如果一个调用返回内存指针，由谁负责释放内存：调用者还是库？如果库负责释放，内存会在何时释放？
- 库调用检查了哪些错误条件，它认为哪些是错误条件？它会如何处理错误？
- 调用的所有返回值（按值传递或按引用传递）是什么？会抛出哪些可能的异常？

针对框架，要记住以下几点：

- 如果继承自一个类，应当调用哪一个构造函数？需要覆盖哪些虚方法？
- 哪些内存要由你负责释放，而哪些内存要由框架负责释放？

#### 理解性能

要知道库或其他代码提供怎样的性能保证，这很重要。即使你的程序不强调性能，也应当确保所用代码在用于特定用途时不会性能太差。例如，一个用于 XML 解析的库可能声称很快，但实际上它会在文件中存储临时信息，这会带来磁盘 I/O 读写，相应地会使性能严重降低。

#### 大 O 记法

程序员通常会用大 O 记法 (big-O notation) 来讨论和说明算法与库的性能。本节将解释算法复杂性分析的一般概念和大 O 记法，但不会提供太多不必要的数学知识。如果你对这些概念已经很熟悉了，可以跳过本节。

大 O 记法指定的是相对性能，而非绝对性能。例如，我们不会说一个算法运行的具体时间（如 300 毫秒），而是会用大 O 记法指出算法的性能随其输入规模的增加将如何变化。排序算法中待排序的项数、基于键查找时散列表中的元素个数，以及要在磁盘间复制的文件的大小，这些都是输入的规模。

注意，大 O 记法只适用于速度依赖于输入的算法。对于无输入或者运行时间不定的算法，大 O 记法并不适用。在实际中，你会发现我们关心的大多数算法的运行时间都依赖于输入，因此这个限制不是大问题。

更正式的理解是：大 O 记法指定了算法运行时间要作为其输入规模的一个函数，这也称为算法的复杂性 (complexity)。不过，实际并没有听上去那么复杂。例如，假设一个排序算法要花 50 毫秒完成 500

个元素的排序，花 100 毫秒完成 1 000 个元素的排序。由于元素个数加倍时，排序花费的时间也加倍，因此这个算法的性能就是其输入的一个线性函数。也就是说，要画出性能相对于输入规模的图时，这就会是一条直线。采用大 O 记法，就能总结如下这个排序算法的性能： $O(n)$ 。这里的 O 只是表示你在使用大 O 记法，而  $n$  表示输入规模。 $O(n)$  指出排序算法的速度是输入规模的一个直接线性函数。

遗憾的是，并非所有算法的性能都随输入规模线性增长。如果真是如此（即所有算法的性能都是输入规模的线性函数），计算机程序运行的速度就会快得多！表 4-1 总结了几种常见的（性能）函数，按性能从好到坏排列。

表 4-1

算法复杂性	大 O 记法	解 释	算 法 举 例
常数	$O(1)$	运行时间独立于输入规模	访问数组中的一个元素
对数	$O(\log n)$	运行时间是输入规模以 2 为底的一个对数函数	使用二分查找寻找有序表中的一个元素
线性	$O(n)$	运行时间相对于输入规模呈比例变化	查找无序表中的一个元素
线性对数	$O(n \log n)$	运行时间是输入规模的一个线性函数乘以它的对数函数	归并排序
平方	$O(n^2)$	运行时间是输入规模的一个平方函数	选择排序之类的较慢排序算法

将性能指定为输入规模的函数而不是绝对的数，这样做有两个优点：

1. 这是平台无关的。指出一段代码在某台计算机上运行时间为 200 毫秒，并不能说明它在另一台计算机上的运行速度如何。如果不基于完全相同的负载在相同的计算机上实际运行两个不同的算法，将很难对这两个算法作出比较。另一方面，性能如果指定为输入规模的函数，那么它就能应用于任何平台。

2. 将性能指定为输入规模的函数，这样只用一个规范就可以涵盖所有的输入。算法运行所需的具体时间（秒数）只是针对一种特定的输入，并不能说明针对其他输入的性能。

#### 理解性能的有关提示

既然已经熟悉了大 O 记法，你就能理解大多数性能说明了。特别地，C++ 标准模板库就使用大 O 记法来描述其算法和数据结构性能。不过，有时大 O 记法可能还不够，甚至会产生误导。在考虑大 O 记法描述的性能规范时，请注意以下问题：

- 如果数据量加倍，算法运行的时间也加倍，这并不能说明这个算法本身的性能好坏！（译者注：虽然采用大 O 记法性能可以写作  $O(n)$ ，但是还可能有其他因素影响实际性能）。如果算法写得很糟糕，即使它能很好地扩展，你可能也不想使用这个算法。例如，假设算法做了不必要的磁盘访问。这也许不会影响大 O 记法，但是实际性能会非常差。
- 如果仅基于大 O 记法，对于采用大 O 记法运行时间相同的算法，将很难做出比较。例如，如果两个不同的排序算法都声称性能为  $O(n \log n)$ ，不实际运行测试的话，就很难区别哪个更快一些。
- 如果输入很少，大 O 记法给出的时间可能会产生误导。对于小的输入规模， $O(n^2)$  算法可能比  $O(n \log n)$  算法表现更好。在做出决定之前，先要考虑可能的输入规模是多大。

除了考虑大 O 记法的特点，还应当了解算法性能的其他方面。以下是需要记住的一些原则：

- 应当考虑使用特定库代码段的频度如何。有些人认为“90/10”规则就很有帮助：大多数程序 90% 的运行时间都只是花在运行 10% 的代码上（Hennessy and Patterson, 2002）。如果你要使用的库代码落在这 10% 经常运行的代码范围内，就一定要仔细分析它的性能特征。另一方面，如果它落在那 90% 不太运行的代码范围内，就不必花太多时间分析它的性能，因为它对程序的整体性能没有太大影响。

- 不要太相信文档。一定要运行性能测试来确定库代码是否提供了可以接受的性能。

#### 理解平台局限性

开始使用库代码之前，要保证已经了解它会在哪些平台上运行。听上去可能很显然。当然，如果一个应用还要在 Linux 上运行，就不要在这个应用中使用 MFC。不过，尽管有些库声称是跨平台的，但在不同平台上它们也可能存在微小的差别。

另外，平台不光包括不同的操作系统，还包括同一操作系统的不同版本。如果你编写了一个要在 Solaris 8、Solaris 9 和 Solaris 10 上运行的应用，要确保使用的所有库也支持前述各种版本。不能自认为操作系统各个版本之间存在向前或向后兼容性。也就是说，一个库能在 Solaris 9 上运行，这并不能说明它还能在 Solaris 10 上运行，反之亦然。Solaris 10 上的库可能会使用这个版本新增的操作系统特性或其他库。另一方面，Solaris 9 上的库使用的某些特性在 Solaris 10 中可能已经删除，或者这些库可能使用了一种古老的二进制格式。

#### 理解许可和支持

使用第三方库通常会带来复杂的许可问题。要使用第三方开发者的库，有时必须向他们缴纳一定的许可费用。而且可能还有其他的许可限制，包括出口限制。另外，开源库往往会在一定的许可条件之下发布，即要求使用这些开源库的代码本身也必须是开源的。

如果你计划发布或出售你开发的代码，一定先要了解所用第三方库的许可限制。如果有疑问，可以咨询一个法律方面的专家。

使用第三方库还会带来支持问题。在使用一个库之前，要确保了解如何提交 bug，而且要知道修正 bug 需要多长时间。如果可能的话，还要明确这个库还有多长的寿命（即开发商在多长时间之内还会对其提供支持），以便做相应的规划。

有意思的是，即使是使用本组织内部的库也可能带来支持问题。你会发现，要想说服本公司另一个部门的一位同事，请他修正他的库中存在的 bug，与说服别家公司里一个素不相识的人相比，难度可能一样大。实际上，你可能会发现说服本公司的人更加困难，因为你并不是一个顾客，没有向他支付任何费用。在使用内部库之前，一定要了解本组织的有关章程条例。

#### 知道从哪里寻求帮助

有时，最初你可能不太敢使用库和框架。幸运的是，你可以从多个渠道获得支持。首先，可以参考库随附的文档。如果这个库得到了广泛使用，如标准模板库 (STL)，或 MFC，你还能找到有关这一主题的不错的参考书。实际上，要得到有关 STL 的帮助，就完全可以参考这本书中的第 21 章～第 23 章。如果有一些特殊的问题，而在书和产品文档中没有相应的解答，可以在网上查查看。在一个诸如 Google (www.google.com) 的搜索引擎上键入问题，找到讨论这个库的相关网页。例如，我在 google 上查 “introduction to C++ STL” 时，就找到了数百个有关 C++ 和 STL 的网站。

请注意：不要完全相信 Web！与公开出版的书和文档相比，网页一般没有经过同样的审查过程，因此可能存在不正确的地方。

还可以考虑浏览新闻组和注册邮件列表。可以搜索 <http://groups.google.com> 的 Usenet 新闻组，了解有关库或框架（译者注：更确切的说，应该是你所用的库或框架）的信息。例如，假设你不知道 C++ 标准中不包括 STL 的散列表。在 google 新闻组中搜索 “hashtable in C++ STL” 时，会发现有很多帖子解释了标准中没有散列表，不过许多开发商以某种方式提供了自己的实现。

新闻组中通常气氛不太好。有些帖子可能很无礼，让人不快，你要根据自己的判断来浏览和发表帖子。

最后一点，许多网站都对特定主题建立了它们自己的专用新闻组，你也可以注册这些新闻组。

### 原型

刚开始接触一个新库或框架时，编写一个快速原型通常是不错的想法。对代码做些试验，这是熟悉库功能的最好的方法。甚至在着手程序设计之前，就应该考虑对库做试验，以便在将其纳入到你的设计之前真正熟悉库的功能和局限性。通过这种经验性的测试，你还能确定库的性能特征。

即使原型应用看上去根本不像是最终的应用，花些时间建立原型绝对不是浪费。不过，不要认为必须为实际应用编写一个原型。可以编写一个“假”程序测试你想用的库功能。目的只是要熟悉这个库。

由于时间限制，程序员有时会发现他们的原型演变成了最终产品。如果你建立了一个原型，而它还不足以作为最终产品的基础，那么一定要避免这种用法。

### 4.1.4 捆绑第三方应用

你的项目可能包括多个应用。也许需要一个 Web 服务器前端来支持新的电子商务基础设施。你的软件可能要捆绑第三方的应用，如一个 Web 服务器。这种方法将代码重用发挥到了极致，因为你重用了整个应用！

不过，使用库的大多警告和原则也同样适用于捆绑第三方的应用。一定要具体了解你所做决定的合法性和许可问题。

将你的软件发布捆绑第三方应用之前，要咨询一个法律方面的专家。

另外，支持问题会变得更加复杂。如果顾客遇到的问题出自你捆绑的 Web 服务器，他们应该与你联系，还是要与 Web 服务器开发商联系呢？在发行软件之前一定要解决这些问题。

### 4.1.5 开源库

开源库是越来越流行的一类可重用代码，开源（open-source）的一般含义是源代码可供任何人查看。如果要在你发布的产品中包括源代码，对此有许多正式的条文和法律，不过对于开源软件，要记住重要的一点是：任何人（包括你）都能查看源代码。需要注意的是，开源并不仅限于开源库。实际上，最著名的开源产品莫过于 Linux 操作系统了。

#### 开源活动

遗憾的是，开源群体中在术语上有些混乱。首先，对于开源活动就同时有两个不同的名字（有人可能会说这是两个单独但相似的活动）。Richard Stallman 和 GNU 项目就使用自由软件（free software）一词。要注意，“自由”（free）这个词并不表示最终完成的产品没有任何费用就一定可用。开发人员也可以为一个自由软件产品交费，或者根据他们的意愿交一点点费用。实际上，自由一词是指人们可以自由地检查源代码、修改源代码以及重新发布这个软件。要把这个词理解为自由谈话中的“自由”；而非免费啤酒中的“免费”。通过访问 [www.gnu.org](http://www.gnu.org) 还能了解到有关 Richard Stallman 和 GNU 项目的更多信息。

不要把自由软件与免费软件混为一谈。免费软件（freeware）或共享软件（shareware）无需费用就可以得到，不过源代码可能是专有的，或私有的（proprietary）。自由软件则不同，可能需要一定费用才能使用，不过源代码肯定可以得到。

开源发起组织 (Open Source Initiative) 使用开源软件 (open-source software) 一词来描述源代码可用的软件。与自由软件一样, 开源软件不要求产品或库一定要免费使用。可以访问 [www.opensource.org](http://www.opensource.org) 来了解有关 Open Source Initiative 的更多信息。

由于“开源”比“自由软件”含义更明确, 这本书将使用“开源”来表示源代码可用的产品和库。选择这个名字并不表示我们认为开源理论优于自由软件理论, 做此选择只是为了便于理解。

#### 查找并使用开源库

不论选用哪一个词 (开源还是自由软件), 通过使用开源软件总能获得很多好处。最主要的好处就是功能。已经有大量可用的开源 C++ 库能够完成各种各样的任务, 从 XML 解析到跨平台的错误日志记录, 都能找到有关的库。

尽管不要求开源库提供免费的发布版本和许可, 但许多开源库确实无需付费就可以拿到。使用开源库的话, 在许可费用这个环节上总能省一些钱。

最后一点, 通常可以自由地修改开源库来满足具体需求。

大多数开源库都可以在网上获得。可以尝试在 google 上查找需要的东西。例如, 如果查找的串是 “open-source C++ library XML parsing”, Google 上首先找到的链接就是一组 C 和 C++ 的 XML 库的链接, 包括 libxml 和 Xerces C++ Parser。

还有一些开源门户网站, 可以在这里搜索你要找的东西, 包括:

- [www.opensource.org](http://www.opensource.org)
- [www.gnu.org](http://www.gnu.org)
- [www.sourceforge.net](http://www.sourceforge.net)

通过自己的搜索, 应该能很快在网上发现更多的资源。

#### 使用开源代码的原则

开源库会带来一些特有的问题, 并且需要新的策略。首先, 开源库通常是人们在其“自由”时间内编写出来的。任何想要投入和参与开发或修正 bug 的程序员一般都可以得到源码基。作为程序设计群体中的一个好公民, 如果你发现自己从开源库中得到了好处, 也应该尝试为开源项目做点贡献。如果你为一家公司工作, 管理层往往不能接受这种想法, 因为这不会为公司带来任何直接收益。不过, 你可以向管理层指出这样做的间接好处, 比如你的公司可能会因此而声名大增, 如果能使公司支持开源活动, 你就能更好地从事有关工作。

其次, 由于开源库开发本身所具有的分布性, 而且缺乏一个具体的所有人, 所以开源库往往会带来支持问题。如果你非常需要修正一个库中的某个 bug, 可以自行修正, 这通常比等着别人来修正效率更高。如果你确实修正了 bug, 就应该把所做的修正放到该库的公共源代码基中。即使你没有修正任何 bug, 也应当报告发现的问题, 这样其他程序员就不会因为遇到同样的问题而浪费时间。

在使用开源库时, 一定要遵从开源活动的“自由”原则。不要滥用这种自由, 或者只是获取而不付出。

#### 4.1.6 C++ 标准库

作为一个 C++ 程序员, 要使用的最重要的库就是 C++ 标准库。顾名思义, 这个库是 C++ 标准的一部分, 因此遵循标准的任何编译器都带有这个库。标准库并不是单独的一个库, 它包括多个分开的组件, 其中有一些你可能已经用到了。甚至可以认为这些库是核心语言的一部分。这一节将从设计的角度介绍标准库中的各个组件。你将了解到可以使用哪些工具, 不过在此你不会看到编写代码的细节。有关的详



细内容将在本书其他章中介绍。

需要注意以下概述并不全面。有些详细内容将在本书后面更合适的地方做讲解，而有些细节则会完全忽略。标准库内容太多，要在一般性的 C++ 书中完整地介绍它不太可能，专门有一些介绍标准库的书，而这些书本身就有 800 页之多！

### C 标准库

由于 C++ 是 C 的一个超集，因此整个 C 库仍然可用。其功能包括数学函数（`abs()`、`sqrt()` 和 `pow()`），利用 `srand()` 和 `rand()` 提供随机数，以及一些错误处理辅助工具（如 `assert()` 和 `errno`）。另外，C 库中有一些将字符数组作为字符串来管理的工具，如 `strlen()` 和 `strcpy()`，还有一些 C 风格的 I/O 函数，如 `printf()` 和 `scanf()`，这些在 C++ 中还能用。

C++ 提供了比 C 更好的字符串和 I/O 支持。尽管在 C++ 中还能使用 C 风格的字符串和 I/O 例程，不过要避免使用这些函数，而应当转而使用 C++ 的字符串和 I/O 流。

我们假设你对 C 库已经很熟悉了。如果不是这样，可以参考附录 B 所列的某一本 C 参考书。还要注意，C 的头文件在 C++ 中文件名会有不同。有关详细内容，请参见网站上的标准库参考（Standard Library Reference）资源。

### 字符串

C++ 提供了内置的 `string` 类。尽管你可能还在使用 C 风格的字符串（字符数组），但不论从哪个方面讲，C++ 的 `string` 类都要更胜出一筹。C++ 的 `string` 类会处理内存管理；提供某种越界检查、赋值语义和比较；而且还支持诸如连接、子串抽取和子串或字符替换等处理。

从理论上讲，C++ `string` 实际上是一个 `typedef` 类型名，这是 `basic_string` 模板的一个 `char` 实例化。不过，无需你操心这些细节；你可以简单地使用 `string`，就好像它是一个真的非模板类一样。

如果忘记了有关内容，可以参考第 1 章，其中回顾了字符串类的功能。网站上的标准库参考（Standard Library Reference）资源则提供了更详细的信息。

### I/O 流

C++ 引入了使用流（stream）完成输入和输出的新模型。C++ 库对于从文件、控制台/键盘和字符串读取内置类型提供了一些标准例程。另外，C++ 还提供了一些工具以便编写自己的例程来读写自己的对象。

第 1 章复习了 I/O 流的基础知识。第 14 章将介绍流的详细内容。

### 国际化

C++ 还提供了对国际化（internationalization）的支持。基于这些特性，可以编写使用不同语言、字符格式和数字格式的程序。

第 14 章将讨论国际化。

### 智能指针

C++ 提供了一个有限的智能指针模板，称为 `auto_ptr`。基于这个模板类，可以包装任何类型的指针，这样当它出了作用域时（译者注：即其工作结束时）就会对所包装的指针自动地调用 `delete`。

不过，这个类并不支持引用计数，因此指针一次只能有一个所有者。

第 13 章、第 15 章、第 16 章和第 25 章将更详细地讨论智能指针。

### 异常

C++ 语言支持异常，从而允许函数或方法将不同类型的错误上传至调用函数或方法（即调用此函数

的上层函数)。C++标准库提供了异常的分类体系，可以在你的程序中加入使用，或者可以派生这些异常来创建自己的异常类型。第15章介绍了异常的详细内容和标准异常类。

### 数学工具

C++库提供了一些数学工具类。尽管这些类都是模板类，以便这些数学工具可以用于任何类型，但一般并不认为这些数学工具类是标准模板库的一部分。除非使用C++来完成数值计算，一般不需要使用这些工具。

标准库提供了复数类，名为 `complex`，它提供了一个抽象，可以处理包含有实部和虚部的复数。

标准库还包含一个名为 `valarray` 的类，从数学角度看，这实际上就是一个向量。标准库提供了一些相关的类来表示向量元素的概念。根据这些构造模块，可以建立完成矩阵处理的类。不过，并没有内置的矩阵类。

C++还提供了一种新的方法来得到有关值域的信息，如当前平台上整数可能的最大值。在C中，可以访问 `#define` 宏来得到这些信息（如 `INT_MAX`），尽管在C++中还可以这样用，不过你还可以使用一组新的 `numeric_limits` 模板类。

### 标准模板库

C++标准库的核心就是它的通用容器和算法。标准库的这个方面通常称为标准模板库（`standard template library`），或者简称为STL，这是因为它大量使用了模板。STL的妙处在于，它以某种方式提供了通用的容器和通用的算法，从而可以在大多数容器中完成大多数算法，而不论容器存储的是何种类型的数据。这一节将介绍STL中的各种容器和算法。第21章到第23章将提供具体的代码来详细说明如何在程序中使用STL。

### STL 容器

STL提供了大多数标准数据结构的实现。在使用C++时，不必再编写诸如链表或队列之类的数据结构。数据结构（或容器，`container`）会以某种方式存储信息片（或元素，`element`），从而能够适当地存取这些元素。不同的数据结构有不同的插入、删除和存取行为和性能特征。一定要熟悉可用的数据结构，这很重要，这样就能为给定的任务选择最合适的数据结构。

STL中的所有容器都是模板，因此可以使用这些容器来存储任何类型，从内置类型（如 `int` 和 `double`）到自己的类都可以。需要注意，任何给定容器中都必须存储相同类型的元素。也就是说，不能在同一个队列中同时存储 `int` 和 `double` 类型的元素。不过，可以分别创建两个单独的队列，一个用于存储 `int`，另一个用来存储 `double`。

C++ STL 容器是同构的：每个容器中只允许有相同类型的元素。

要注意，C++标准指定了每个容器和算法的接口，而非实现。因此，不同开发商完全可以提供不同的实现。不过，标准还将性能需求指定为接口的一部分，实现必须满足这些性能需求。

本节将提供STL中各种可用容器的概述。

### 向量

向量（`vector`）会存储一个元素序列，并提供对这些元素的随机访问。可以把向量看作是一个元素数组，当插入元素时它会动态扩展，而且提供了某种越界检查。类似于数组，向量的元素也存储在连续的内存空间中。

C++中的向量是动态数组的同义词：随着所存储元素个数的变化，向量会动态地扩展和收缩。C++ `vector` 并不是指数学意义上的向量概念。C++ 将数学意义上的向量建模为 `valarray` 容器。

对向量插入和删除元素时，如果在向量最后位置进行插入和删除，速度会很快（常量时间），但是在别处插入和删除时速度就较慢（线性时间）。此时插入和删除之所以比较慢，是因为操作必须将所有元素“下移”或“上移”一个位置，以便为新元素留出空间（插入操作），或者填充所删除元素原来占的空间（删除操作）。类似于数组，向量提供了快速（常量时间）的元素存取，即可以在常量时间内存取任何元素。

如果需要快速存取元素，但不打算经常增加或删除元素，就应当在程序中使用向量。有一个很好的经验，即只要使用数组的地方都可以用向量。例如，系统监控工具可能会把它所监控的计算机系统列表保存在一个向量中。很少会向这个列表中增加新的计算机，也很少从这个列表中删除当前的计算机。不过，用户可能经常希望查找有关一台特定计算机的信息，因此查找时间应当很快才行。

应当尽可能使用向量，而不是数组。

### 列表

STL 列表（list）是标准的链表结构。类似于数组或向量，它也存储了一个元素序列。不过，与数组或向量不同，链表的元素不一定存储在连续的内存空间中。相反，列表中的每个元素指定了在哪里寻找列表中的下一个和前一个元素（通常通过指针来实现）。需要注意，如果列表中的元素同时指出了下一个和前一个元素，这种列表就称为双向链表（doubly linked list）。

列表的性能特征与向量恰好相反。列表提供了较慢（线性时间）的元素查找和存取，但是一旦找到相关的位置，完成元素的插入和删除则很快（常量时间）。因此，如果打算插入和删除很多元素，但是不要求查找很快，就应当使用一个列表。例如，在聊天室实现中，可以将当前参与聊天的所有人存储在一个列表中。聊天室里的人可能会频繁地进来出去，所以需要快速的插入和删除。另一方面，很少需要查看列表中的人员，所以即使查找时间较慢，你也不会太在意。

### 双端队列

双端队列（deque）是 double-ended queue 的简写。双端队列是介于向量和列表之间的中间产物，不过更接近于向量。与向量类似，它提供了快速（常量时间）的元素存取。另外，类似于列表，在序列两端插入和删除时，速度也很快（摊分常量时间）。不过，与列表不同的是，在序列中间插入和删除时，速度则较慢（线性时间）。

如果需从序列的两端插入或删除，但仍需快速地存取所有元素，就应当使用双端队列而不是向量。不过，这个需求对于许多编程问题都不适用。在大多数情况下，向量或队列应该就足够了。

向量、列表和双端队列也称为顺序容器（sequential container），因为它们都存储了一个元素序列。

### 队列

队列（queue）一词实际上取自英语中对队列的定义，这表示一队人或一队对象。队列容器提供了标准的先进先出（first in, first out，或 FIFO）语义。队列作为一个容器，你可以在其一端插入元素，而在另一端将元素取出。元素的插入和删除都很快（常量时间）。

如果想对实际生活中的“先来先服务”语义建模，就应当使用一个队列结构。例如，可以考虑一个银行。顾客到达银行时，他们要排队。出纳完成了第一个顾客的业务后，会为排队的下一个顾客提供服务，这样就提供了一种“先来先服务”的行为。通过把 Customer 对象存储在一个队列中，可以实现一个银行仿真。每个顾客到达银行时，把他或她增加到队尾。出纳要为顾客提供服务时，处在队头的顾客就会得到服务。如此一来，顾客就会按其到达的顺序获得服务。

### 优先队列

优先队列 (priority queue) 提供了一种队列功能, 其中每个元素都有一个优先级。元素会按优先级顺序从队列中删除。如果优先级相同, 则仍然遵循 FIFO 语义, 即第一个插入的元素会先删除。优先队列的插入和删除一般都比简单队列的插入和删除慢, 因为必须对元素重新调整顺序, 以支持按优先级排序。

可以使用优先队列对“超常队列”建模。例如, 在上述银行仿真中, 假设有企业账户的顾客比一般的顾客优先级高。实际生活中的许多银行都会排两个队来实现这种行为: 一个是企业顾客的队, 另一个是其他顾客的队。企业队列中的顾客会在另一个队中的顾客之前得到服务。不过, 银行用一个队也可以提供这种行为, 只需要简单地将企业顾客移到所有非企业顾客的前面去。在你的程序中, 可以使用一个优先队列, 其中顾客的优先级要么是企业优先级, 要么是一般优先级。所有企业顾客都应当在所有一般顾客之前得到服务, 但是在优先级相同的同一组顾客中, 仍然按先来先服务的顺序提供服务。

### 栈

STL 的栈 (stack) 提供了标准的先进后出 (first-in, last-out 或 FILO) 语义。与队列相似, 会向这个队列中插入和删除元素。不过, 在栈中, 最近插入的元素会最先删除。栈这个名字得自于这个结构的一个可视化显示, 即在一堆对象中, 只能看到最上面的一个对象 (译者注, 栈也称为堆栈)。向栈中增加对象时, 就会盖住 (隐藏) 在它之下的所有对象。

栈是对实际生活中的“先来后服务”行为建模。举例来说, 可以考虑一个大城市的停车场, 先到的车会被后来的车“堵”在里面。在这种情况下, 最后来的车反而能够最早离开。

STL 栈容器提供了元素的快速 (常量时间) 插入和删除。如果想得到 FILO 语义, 应当使用栈结构。例如, 一个错误处理工具可能希望把错误存储在一个栈中, 这样最后出现的错误能够最早让管理员看到。按 FILO 顺序处理错误通常很有用, 因为新的错误有时会使老错误失去意义。

从技术上讲, 队列、优先队列和栈容器都是容器适配器 (container adapter)。它们是建立在三种标准顺序容器 (向量、双端队列和列表) 之上的接口。

### 集合和多集

STL 中的集合 (set) 就是一个元素集合 (collection)。尽管集合的数学定义指出它是无序的, 但是 STL 的集合会按有序的方式存储元素, 这样它就能提供相当快的查找、插入和删除。事实上, 集合的插入、删除和查找性能都是对数函数, 这比向量提供的插入和删除快, 并且比列表提供的查找要快。不过, 与列表的插入和删除相比会慢一些, 另外查找则比向量的查找慢。其底层实现往往是一个平衡二叉树, 如果你平常要使用平衡二叉树结构, 就应当使用集合。具体地, 如果插入/删除和查找一样多, 而且希望尽可能地优化, 集合就很适用。例如, 在一个业务繁忙的书店里, 书目记录程序可能就会使用一个集合来存储图书。只要有新书来或者有书卖出, 店内的书目列表就必须得到更新, 因此插入和删除必须很快。顾客还需要能够查找某一本特定的书, 因此这个程序还应当提供快速的查找。

如果希望插入、删除和查找的性能相当, 则应当使用集合, 而不是向量或列表。

需要注意的是, 集合中不允许有重复的元素。也就是说, 集合中的每个元素必须惟一。如果你希望存储重复的元素, 就必须使用一个多集 (multiset)。

多集只不过是一个允许元素重复的集合。

### 映射和多映射

映射 (map) 存储了键/值对。元素按键排序。在其他各个方面, 它与集合完全相同。如果想要建立键和值的关联, 就应当使用一个映射。例如, 在一个在线的多人游戏中, 你可能希望存储有关各个玩家的一些信息, 如他或她的登录名、实际姓名、IP 地址和其他特征。可以把这个信息存储在一个映射中, 并使用玩家的登录名作为键。

多映射 (multimap) 与映射的关系就相当于多集与集合之间的关系。具体地, 多映射就是允许有重复键的映射。

需要注意的是, 可以将映射作为一个关联数组 (associative array)。也就是说, 可以把它作为一个数组, 其中索引可以是任何类型, 如可以是一个字符串。

集合和映射容器也称为关联容器 (associative container), 因为它们建立了键与值的关联。应用于集合时, 这个词有些让人困惑, 因为集合中, 键本身就是值。由于这些容器会对其元素排序, 所以也称为有序 (sorted) 关联容器。

### 位集

C 和 C++ 程序员经常会在一个 int 或 long 中存储一组标志, 每个标志用一位表示。他们使用位操作符 (&、|、~、~、<<、和 >>) 来设置和访问这些位。C++ 标准库则提供了一个 bitset 类来抽象这种位操作, 所以不用再使用位处理操作符了。

位集 (bitset) 容器并不是一般意义上的容器, 因为它没有实现一种特定的数据结构以供插入和删除元素。不过, 可以把它看作是可读写的布尔值序列。

### STL 容器小结

表 4-2 对 STL 提供的容器做了一个总结。在此使用大 O 记法来表示包含 N 个元素的容器的性能特征。表中的 N/A 项表示: 相应操作不是容器语义中的一部分。

表 4-2

容器类名	容器类型	插入性能	删除性能	查找性能	何时使用
vector (向量)	顺序容器	在最后位置插入时性能为 $O(1)$ , 在其他位置插入时性能为 $O(N)$	在最后位置删除时性能为 $O(1)$ , 在其他位置删除时性能为 $O(N)$	$O(1)$	需要快速查找, 不在意插入/删除的速度慢, 就应当使用向量 能使用数组的地方都能使用向量
list (列表)	顺序容器	$O(1)$	$O(1)$	$O(N)$	需要快速的插入/删除 不在意查找的速度慢, 就应当使用列表
deque (双端队列)	顺序容器	在开始或最后位置插入时性能为 $O(1)$ , 在其他位置插入时性能为 $O(N)$	在开始或最后位置删除时性能为 $O(1)$ , 在其他位置删除时性能为 $O(N)$	$O(1)$	一般不需要双端队列; 可以转而使用一个 vector 或 list
queue (队列)	容器适配器	$O(1)$	$O(1)$	N/A	需要一个 FIFO 结构时就可以使用队列
priority_queue (优先队列)	容器适配器	$O(\log(N))$	$O(\log(N))$	N/A	需要一个带优先级的 FIFO 结构时就可以使用优先队列
stack (栈)	容器适配器	$O(1)$	$O(1)$	N/A	需要一个 FILO 结构时就可以使用栈

(续)

容器类名	容器类型	插入性能	删除性能	查找性能	何时使用
set / multiset (集合/多集)	有序关联 容器	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$	需要一个元素集合， 而且对元素的查找、插 入和删除同样多，就可 以使用集合/多集
map / multimap (映射/多映射)	有序关联 容器	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$	希望将键与值相关联， 就可以使用映射/多映射
bitset (位集)	特殊容器	N/A	N/A	$O(1)$	需要一个标志集合时 就可以使用位集

需要注意，string 从技术上讲也是容器。可以认为它们是字符的向量 (vector)。因此，以下介绍的一些算法同样适用于 string。

### STL 算法

除了容器外，STL 还提供了许多通用算法的实现。算法 (algorithm) 就是一种完成特定任务的策略，如排序或查找。这些算法也实现为模板，因此在大多数不同的容器类型上都能使用。需要注意，算法一般不是容器的一部分。STL 采用了一种不寻常的方法，即把数据 (容器) 与功能 (算法) 分离。尽管这种方法看上去与面向对象程序设计的精神有所违背，但为了支持 STL 中的通用程序设计，这是必要的。正交性 (orthogonality) 指导原则就要求算法和容器是独立的，(几乎) 任何算法都能用于 (几乎) 任何容器。

尽管算法和容器理论上是独立的，但是有些容器以类方法的形式提供了某些算法，因为对于这些特定容器来说，通用算法的表现可能不太好。例如，集合提供了自己的 find() 算法，它比比通用的 find() 算法要快。如果提供了方法形式的算法，就应当使用这样的算法，因为通常它的效率更高，或者更适合于当前容器。

需要注意，算法并非直接在容器上工作。它们使用了一个“中间人”，称之为迭代器 (iterator)。STL 中的每个容器都提供了一个迭代器，它会把容器中的元素遍历到一个序列中。即使是集合和映射中的元素，迭代器也会临时地将这些容器中的元素转换到序列中。对应各种容器的不同迭代器都遵循标准接口，因此算法可以使用迭代器完成工作，而不必操心底层的容器实现。要了解有关迭代器、算法和容器的所有详细内容，可以参考第 21 章~第 23 章以及网站上的资料。

迭代器作为算法和容器之间的中间人，它们提供了一个标准接口，可以遍历访问容器中的元素，并置于一个序列中，这样任何算法都可以在任何容器上工作。后面将进一步讨论迭代器设计模式。

STL 中大约有 60 种算法 (要看你怎么来数了)，通常可以划分为几个不同的大类。不同书中在此分类上可能稍有不同。这本书中认为所有 STL 算法可以分为 5 类：工具算法、非修改算法、修改算法、排序算法和集合算法。有些类还可以进一步细分。要注意，如果以下算法指定为在一个元素“序列”上工作，那么该序列就会作为一个迭代器提供给算法。

在查看以下算法表时，要记住一点，STL 是由一个委员会设计的。援引一句玩笑话：“斑马就是委员会设计马时得到的杰作。”换句话说，委员会所得到的设计中通常会包含一些额外的或不需要的功能，如非要为马加上斑马的条纹。你可能会发现，STL 中的某些算法同样很奇怪，或者是不必要的。确实如此。你不必使用 STL 提供的每一个算法。重要的是，在需要某个算法的时候只要知道有这样一个算法可用就行了。



### 工具算法

与其他算法不同，工具算法不在数据序列上工作。之所以认为它们也是 STL 的一部分，只是因为它们是模板化的算法。

算 法 名	算 法 说 明
min(), max()	返回两个值中的最小值或最大值
swap()	交换两个值

### 非修改算法

所谓非修改算法是指，这些算法只查看一个元素序列，并返回有关元素的某个信息，或者在各元素上执行某个函数。既然是“非修改”算法，它们不会修改元素的值，也不会改变序列中元素的顺序。这一类中包括 4 种算法。下面的各表列出了各种非修改算法，并提供了简要的总结。利用这些算法，应该很少需要编写 for 循环来对一个值序列进行迭代处理了。

#### 查找算法

算 法 名	算 法 说 明	是否需要有序序列
find(), find_if()	查找与某个值匹配的的第一个元素，或者导致一个谓词返回 true 的第一个元素	不需要
find_first_of()	类似于 find，不过会同时搜索多个元素中的某一个	不需要
adjacent_find()	查找彼此相等的两个连续元素的第一个实例	不需要
search(), find_end()	查找序列中与另一个序列相匹配的第一个子序列 (search()) 或最后一个子序列 (find_end())	不需要
search_n()	查找等于一个给定值的 n 个连续元素的第一个实例	不需要
lower_bound(), upper_bound(), equal_range()	查找包括一个特定元素的范围的开始位置 (即下界, lower_bound()), 结束位置 (即上界, upper_bound()), 或者上下界 (equal_range())	需要
binary_search()	在一个有序序列中查找一个值	需要
min_element(), max_element()	在一个序列中查找最小或最大元素	不需要

### 数值处理算法

算 法 名	算 法 说 明
count(), count_if()	统计与一个值匹配或者导致某个谓词返回 true 的元素个数
accumulate()	“累计”序列中所有元素的值。默认行为是对元素求和，不过调用者可以提供不同的二元函数
inner_product()	类似于累计 (accumulate())，但在两个序列上工作。它会基于两个序列中相对应的两个元素 (并行元素) 调用一个二元函数，并将结果累加。默认的二元函数是乘法，如果序列表示的是数学意义上的向量，那么此算法就会计算向量的点乘
partial_sum()	生成一个新序列，其中每个元素都是源序列中相应位置上的元素及前面所有元素的和 (或其他二元操作)
adjacent_difference()	生成一个新序列，其中每个元素都是源序列中相应位置上的元素与其前一个元素的差

### 比较算法

算 法 名	算 法 说 明
<code>equal()</code>	通过检查两个序列的元素顺序是否相同（即相应位置上的元素是否相同），从而确定两个序列是否相等
<code>mismatch()</code>	返回各序列中与另一个序列同一位置上的元素不匹配的第一个元素
<code>lexicographical_compare()</code>	比较两个序列，确定它们的“字典”顺序。将第一个序列中的各个元素与第二个序列中相应的元素进行比较。如果一个元素小于另一个元素，那么前一个序列就先于后一个序列。如果两个元素相等，则按顺序比较下一对元素。

### 运算算法

算 法 名	算 法 说 明
<code>for_each()</code>	对序列中的每个元素执行一个函数。这个算法对于打印容器中的各个元素很有用

### 修改算法

修改算法会修改序列中的某些或全部元素。有些算法会就地（in place）修改元素，因此源序列会改变。其他一些算法则会把结果复制到另一个序列中，因此源序列保持不变。下表总结了修改算法。

算 法 名	算 法 说 明
<code>transform()</code>	在每个元素或每对元素上调用一个函数
<code>copy()</code> 、 <code>copy_backward()</code>	将元素从一个序列复制到另一个序列
<code>iter_swap()</code> 、 <code>swap_ranges()</code>	交换两个元素或两个元素序列
<code>replace()</code> 、 <code>replace_if()</code> 、 <code>replace_copy()</code> 、 <code>replace_copy_if()</code>	将与某个值匹配或者导致一个谓词返回 true 的所有元素替换为一个新元素，可以就地替换，也可以将结果复制到一个新序列中
<code>fill()</code> 、 <code>fill_n()</code>	将序列中的所有元素设置为一个新值
<code>generate()</code> 、 <code>generate_n()</code>	类似于 <code>fill()</code> 和 <code>fill_n()</code> ，不过会调用一个指定的函数来生成放在序列中的值
<code>remove()</code> 、 <code>remove_if()</code> 、 <code>remove_copy()</code> 、 <code>remove_copy_if()</code>	从序列中删除与某个给定值匹配或导致一个谓词返回 true 的元素，可以就地完成，也可以将结果复制到另一个序列中
<code>unique()</code> 、 <code>unique_copy()</code>	从序列中删除连续的重复出现，可以就地完成，也可以将结果复制到另一个序列中
<code>reverse()</code> 、 <code>reverse_copy()</code>	将序列中的元素逆序放置，可以就地完成，也可以将结果复制到另一个序列中
<code>rotate()</code> 、 <code>rotate_copy()</code>	交换序列的前一半和后一半，可以就地完成，也可以将结果复制到另一个序列中。交换的两个子序列的大小不必相同
<code>next_permutation()</code> 、 <code>prev_permutation()</code>	将序列转换为“下一个”或“前一个”排列，从而修改序列。连续地调用某个排列函数（一直调用 <code>next_permutation()</code> ，或一直调用 <code>prev_permutation()</code> ），就可以得到元素的所有可能的排列

### 排序算法

排序算法是一类特殊的修改算法，它会对序列中的元素进行排序。STL 提供了多种不同的排序算法，其性能保证也有所不同。

算 法 名	算 法 说 明
sort()、stable_sort()	就地对元素排序，原先重复元素的先后顺序可能依然保持，也可能有变化（译者注：这就是排序算法的稳定性概念，如果能保持重复元素的先后顺序，则称算法具有稳定性，否则就是非稳定的）。sort() 的性能类似于快速排序，stable_sort() 的性能类似于归并排序（不过具体算法可能存在差别）
partial_sort()、partial_sort_copy()	对序列进行部分排序：前 n 个元素会排序，余下的元素则未排序。可以就地完成，也可以将结果复制到一个新序列中
nth_element()	重新查找序列的第 n 个元素，就好像整个序列已经排序一样
merge()、inplace_merge()	合并两个有序序列，可以就地完成，也可以将结果复制到一个新序列中
make_heap()、push_heap()、pop_heap()、sort_heap()	堆是一种标准数据结构，数组或序列中的元素在堆中以一个半有序的方式排列，因此要找到“堆顶”元素是很快。利用这 4 个算法，可以对序列使用堆排序（译者注：这里的堆是指一种可以与完全二叉树对照考虑的序列结构，而堆顶就是序列中的最小或最大元素）
partition()、stable_partition()	对序列排序，从而让某个谓词返回 true 的所有元素都出现在让该谓词返回 false 的元素之前，各部分中原来的元素顺序可能保持，也可能不保持
random_shuffle()	随机重排元素，将序列置为无序（unsort）

### 集合算法

集合算法是一种特殊的修改算法，即在序列上完成集合操作。这些算法最适于处理来自 set 容器的序列，不过，对于来自大多数容器的有序序列也可以使用集合算法。

算 法 名	算 法 说 明
includes()	确定一个序列是否是另一个序列的子集
set_union()、set_intersection()、set_difference()、set_symmetric_difference()	对两个有序序列完成指定的集合操作，将结果复制到第三个有序序列中。有关集合操作的解释请参见第 22 章

### 选择一个算法

有这么多的算法，而且算法的功能也这么多，刚开始时你可能会被吓住。乍一看确实很难了解如何使用这些算法。不过，既然已经熟悉了有哪些可用的选择，应该能更好地完成程序设计了。要了解如何在代码中使用这些算法，第 21 章～第 23 章会介绍有关的详细内容。

### STL 还少些什么

STL 非常强大，但是它也不是十全十美的。以下列出的是 STL 缺少和未支持的功能：

- STL 不支持同步来保证多线程安全。STL 标准未提供任何多线程同步支持，因为线程是依赖于平台的。因此，如果你有一个多线程程序，就必须为容器实现自己的同步。
- STL 不支持散列表。更一般地说，它不支持任何散列关联容器，在这种关联容器中，元素不是有序的，而是根据散列方法来存储。注意，STL 的许多实现提供了一个散列表或散列映射，但是由于它不是标准的一部分，使用这样的实现不是可移植的。第 23 章提供了一个示例散列映射实现。
- STL 不支持任何通用的树或图结构。尽管映射和集合一般都实现为平衡二叉树，但 STL 并未在接口中对外提供这种实现。如果需要一个树或图结构来完成像编写解析器之类的工作，就需要实现自己的树或图结构，或者在另一个库中找到这样的一个实现。
- STL 不支持任何表抽象。如果想实现棋盘之类的东西，可能需要使用一个二维数组。

不过，要记住重要的一点，STL 是可扩展的（extensible）。可以编写自己的容器或算法，它们能够与既有的算法或容器一同工作。因此，如果 STL 没有提供你所需要的东西，可以考虑自行编写所需的代码，让它与 STL 共同达成目的。

### 决定是否使用 STL

设计 STL 时把功能、性能和正交性摆在了优先的位置上。它的设计并没有太多地考虑易用性，因此，很自然地，最后就表现得不那么容易使用。实际上，本章的介绍还只是触及到 STL 复杂性的皮毛而已。因此，在学习如何使用 STL 时，存在一个很陡的学习曲线。不过，STL 的优点也是显著的。你可以考虑一下，原先在对付链表或平衡二叉树实现中的指针错误时花了多少时间，或者对一个未能正确排序的排序算法进行调试时耗费了多少精力。但如果正确地使用了 STL，就很少需要（甚至不需要）再完成这样的编码工作了。

如果决定在程序中采用 STL，可以参考第 21 章～第 23 章。这几章对于如果使用 STL 提供的容器和算法提供了深入的介绍。

## 4.2 利用模式和技术完成设计

学习 C++ 语言和成为一个好的 C++ 程序员完全是两码事。如果坐下来阅读 C++ 标准，把每一条都记下来，那么你对 C++ 的了解与别人并没有两样。不过，如果不查看代码并编写自己的程序，由此来获得一些经验，你肯定成不了一个好的程序员。原因在于，C++ 语法只是以最原始的方式定义了这种语言能够做什么，而没有指出每个特性应该如何使用。

随着 C++ 程序员积累了越来越多使用 C++ 语言的经验，他们会对使用该语言的特性建立起自己的一些方法。C++ 群体作为一个整体也建立了一些标准方法，以便充分地利用 C++ 语言，其中有些方法是正式的，有些则是非正式的。在本书中，我们指出了这些可重用的语言应用，这也称为设计技术（design technique）和设计模式（design pattern）。另外，第 25 章和第 26 章对设计技术和模式做了比较完备的介绍。有些模式和技术可能是显而易见的，因为它们只是形式化地表述了一个显然的解决方案。其他一些模式则描述了全新的解决方案，可用以解决以前所遇到的问题。

一定要熟悉这些模式和技术，这很重要，由此可以了解到在什么情况下需要利用其中的某个解决方案来解决特定的设计问题。除了本书中介绍的方法外，还有更多可应用于 C++ 的技术和模式。尽管作者认为这本书已经介绍了最有用的一些模式，不过你可能还希望参考一本有关设计模式的书，来了解更多的不同模式和技术。请参见附录 B 建议的参考书目。

### 4.2.1 设计技术

设计技术只是一种用于解决 C++ 中某个特定问题的标准方法。通常，设计技术的目的是为了克服一个不好的特性，或者解决 C++ 中存在的语言缺陷。在另外一些情况下，设计技术就是一段代码，可以在多个不同程序中用来解决一个常见的问题。

#### 设计技术示例：智能指针

C++ 中的内存管理经常会带来错误和 bug。其中许多 bug 都是因为使用动态内存分配和指针造成的。如果在程序中大量使用动态内存分配，并在对象之间传递多个指针，此时往往很难记住要对每个指针调用一次（且仅一次）delete。如果没有这样做，后果是很严重的。如果把动态分配的内存释放了多次，就可能导致内存破坏，如果忘记释放动态分配的内存，又会带来内存泄漏。

智能指针（Smart pointer）可以帮助管理动态分配的内存。从概念上讲，智能指针就是动态分配的内存的一个指针，当该内存出作用域后（译者注：即其工作结束后），这个指针会记住要将此内存释放掉。

在程序中，智能指针通常就是一个对象，其中包含一个常规指针（或哑指针）。这个对象在栈上分配。当对象出作用域后，它的析构函数会在所包含的指针上调用 delete。

注意，有些语言实现提供了垃圾回收（garbage collection）机制，这样程序员不必负责释放任何内存。在这样一些语言中，所有指针都可以认为是智能指针，因为无需记住释放指针所指的任何内存。有些语言（如 Java）提供了垃圾回收作为常规做法，但是相比之下，对于 C++ 来说，编写一个垃圾回收器相当困难。因此，智能指针只是一种补救技术，用以弥补 C++ 未在内存管理中提供垃圾回收的这一缺陷。

管理指针时，除了要记住指针出作用域后应撤销（释放）指针，还存在很多其他问题。有时，多个对象或多个代码段包含有同一个指针的多个副本。这个问题称为别名（aliasing）。为了适当地释放所有内存，应当由使用内存的最后一段代码对指针调用 delete。不过，通常很难知道哪一段代码最后使用内存。甚至确定代码的先后顺序也是不可能的，因为这可能取决于运行时的输入。因此，还有一种更复杂的智能指针，它实现了引用计数（reference counting）来跟踪其所有者。如果所有所有者都使用完了指针，引用数就会减至 0，智能指针就会对其底层哑指针调用 delete。许多 C++ 框架都大量使用了引用计数，如 Microsoft 的对象链接和嵌入（Object Linking and Embedding, OLE）和组件对象模型（Component Object Model, COM）。即使你不想自己实现引用计数，也应当熟悉这些概念，这是很重要的。

C++ 提供了许多语言特性，使得智能指针更有魅力。首先，可以为所有使用模板的指针类型编写一个类型安全（typesafe）的智能指针类。其次，可以使用操作符重载为智能指针对象提供一个接口，从而允许代码像使用哑指针一样地使用智能指针对象。具体地，可以重载 \* 和 -> 操作符，这样客户代码就可以像对正常指针解除引用一样，对一个智能指针对象解除引用。第 25 章提供了一个带引用计数的智能指针实现，可以把它作为插件直接在程序中使用。C++ 标准库还提供了一个简单的智能指针，称为 auto\_ptr，这在标准库的概述中已经介绍过。

#### 4.2.2 设计模式

设计模式（design pattern）是用以解决一般问题的组织程序的标准方法。C++ 是一种面向对象语言，因此 C++ 程序员感兴趣的设计模式通常都是面向对象模式，指出了在程序中组织对象和对象关系的一些策略。这些模式通常可以应用于任何面向对象语言，如 C++、Java 或 Smalltalk。实际上，如果你对 Java 程序设计很熟悉，就会在其中发现这里的很多模式。

相对于设计技术来说，设计模式与具体语言的关系不大。模式和技术之间的区别很模糊，不同的书可能会有不同的定义。本书中把技术定义为一种特定于 C++ 语言的策略，用于克服语言本身存在的某个缺陷。而模式则是一种有关面向对象设计的更一般的策略，可以应用于任何面向对象语言。

需要注意，许多模式都有多个不同的名字。模式本身之间的区别也有些含糊，不同的资料对它们的描述和分析可能会稍有不同。实际上，取决于你用的书或其他资源，可能发现不同的模式还会有同样的名字。对于哪些设计方法可以作为模式，这一点更是未达成一致。除了少许例外，本书都采用《Design Patterns: Elements of Reusable Object-Oriented Software》一书中的术语，这是 Erich Gamma 等人所著的一本经典著作。不过，在适当的时候，本书还会指出其他的一些模式名和变种。

第 26 章提供了不同设计模式的一个名录，其中还包括 C++ 示例实现。

##### 设计模式示例：迭代器

迭代器模式提供了这样一种机制，可以将算法或操作与其处理的数据相分离。乍一看，这种模式似乎与面向对象程序设计中的一个基本原则相违背，在面向对象程序设计中，要求将对象数据与处理数据的行为组合在一起。尽管在某个层次上看，这个观点是对的，但是需要指出，这种模式并不是提倡将基

本行为从对象中去除。相反，它解决了数据与行为紧耦合时通常出现的两个问题。

将数据和行为紧耦合时，第一个问题是：这会阻碍通用算法的编写和使用，这些算法可以处理多种对象，而所处理的对象并非都在同一个类层次体系中。为了编写通用算法，往往需要某种标准机制来访问对象的内容。

数据与行为紧耦合时的第二个问题是：有时很难增加新的行为。至少，需要访问数据对象的源代码。不过，如果要调整的对象层次体系是一个第三方框架或库的一部分，不允许修改，又当如何呢？如果能增加一个处理数据的算法或操作，而不用修改原来的数据对象层次体系就好了。

你已经看过 STL 中的一个迭代器模式的例子。从概念上讲，迭代器提供了一种机制，允许操作或算法访问一个容器中的元素并置于一个序列中。迭代器这个名字得自于英语单词 *iterate*（迭代），它的意思就是“重复”。由于迭代器要重复做一个动作，在序列中前移从而到达每一个新元素，因此这个词很贴切。在 STL 中，通用算法就使用迭代器来访问其操作的容器中的元素。STL 定义了一个标准迭代器接口，允许编写能够在任何容器上工作的算法，只要该容器提供了一个有适当接口的迭代器就可以。因此，利用迭代器，就能编写通用算法，而无须修改数据。图 4-1 把迭代器显示为一条装配线，它把数据对象元素发送给一个“操作”。

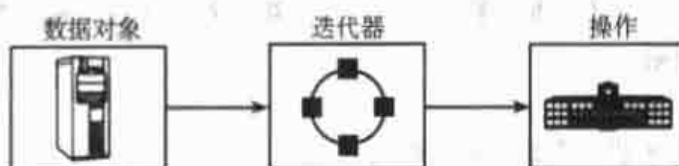


图 4-1

### 4.3 小结

本章强调了重用这一设计原则。你应当了解到，C++ 设计应当包括两类重用：代码重用和思想重用，代码重用的形式是库和框架，而思想重用的形式是技术和模式。

尽管代码重用是一个总目标，但是有关代码重用既有优点也有缺点。在本章中，你了解了重用代码的这些权衡考虑，以及有关的特定原则，包括理解功能和局限性、性能、许可和支持模型、平台限制、原型，以及从哪里寻求帮助。你还了解了性能分析和大 O 记法，以及使用开源库时存在的一些特殊问题和考虑。

本章还对 C++ 标准库提供了一个概述，这也是在编写代码时使用的最重要的一个库。这个库包容了 C 库，并为字符串、I/O、错误处理和其他任务加入了另外一些工具。其中还包括一些通用容器和算法，这些则统称为标准模板库。第 21 章~第 23 章将详细介绍标准模板库。

在设计程序时，重用模式和技术与重用代码同样重要。应当避免重新构建，还要避免重蹈覆辙！最后，这一章介绍了设计技术和模式的概念。第 25 章~第 26 章提供了另外一些例子，包括技术和模式的代码和示例应用。

不过，使用库和模式只是重用策略中的一部分。还需要适当地设计（design）自己的代码，以便你和其他人尽可能地重用。第 5 章将介绍设计可重用代码的策略。



## 第5章 重用设计

要在程序中重用库和其他代码，这是一个重要的设计策略。不过，这只是重用策略的前一半。另一半则是设计和编写可以在程序中重用的代码。你可能已经发现了，在设计得好和设计得不好的库之间存在显著的差异。设计得好的库很好使用，而设计不好的库可能会让你心生嫌恶，以至于干脆放弃，自己动手来编写代码。不论你是在编写一个专门设计要由其他程序员使用的库，还是仅仅确定一个类层次体系，设计代码时都应该把重用谨记心头。你无法知道在以后的某个项目中是否还需要一个同样的功能。

第2章介绍了重用的设计原则。第4章解释了如何在设计中结合库和其他代码来应用这种原则。本章则讨论重用的另一面：如何设计可重用的代码。这里以第3章所述的面向对象设计原则为基础，还将介绍一些新的策略和指导原则。

读完本章之后，你将了解到：

- 重用方法论：为什么要设计可供重用的代码
- 如何设计可重用的代码
  - 如何使用抽象
  - 建立代码以供重用的三种策略
  - 设计可用接口的六种策略
- 如何协调一般性和易用性

### 5.1 重用方法论

应当把代码设计为你和其他程序员都能重用。这条原则不仅适用于你特意希望其他程序员使用的库和框架，还适用于为程序所设计的任何类、子系统或组件。一定要谨记这样一条座右铭：“编写一次，经常使用”。采用这种设计方法有许多原因：

- 代码很少只用于一个程序中。在写代码时，你可能没有打算这些代码会得到重用，不过几个月或者几年之后，你会发现自己或者你的同事可能在一些类似的项目中采用了这个程序的一些组件。由此可以了解到，代码可能会以某种方式再次得到使用，所以开始的时候就应该考虑充分，正确地做出设计。
- 设计可重用的代码能够节省时间和金钱。如果设计代码时根本不考虑将来的使用，以后当遇到需要实现一种类似功能的时候，你或你的同事肯定要花时间再做一次同样（或类似）的设计。即使没有完全杜绝重用，但如果你提供的接口不好，或者缺少功能，将来使用这段代码时就需要花费额外的时间和精力。
- 同组的其他程序员必须能够使用你所写的代码。你的代码可能只适用于当前的特定程序，即使在这样一些情况下，你有可能并非孤身作战来完成项目。如果你花时间为你的同事提供了设计得当、功能完备的库和代码段以便其使用，他们肯定会非常感激的。你很清楚，如果别人写了不好的接口或者欠考虑的类，在使用这些接口和类时是多么的苦恼。设计可重用的代码还可以称为协

作编码 (cooperative coding)。应当适当地编写代码, 使得除了当前项目中的程序员可以由此得到好处, 其他项目 (包括将来项目) 中的程序员也能由此获益。

- 你将成为自己所做工作的最大受益人。有经验的程序员从来不会把代码扔掉。经过一段时间以后, 他们就能建立起自己的一个发展工具库。你无法知道将来是否还需要同样的功能。例如, 本书作者之一最早是在上研究生的时候学过网络编程课程, 他写了一些通用网络例程来创建连接、发送消息和接收消息。从那以后, 每次开发涉及到网络的项目时他都会参考这些代码, 而且在许多不同的程序中重用了其中的一些代码段。

如果你作为公司员工设计或编写了一些代码, 那么拥有知识产权的是你的公司, 而不是你。如果你已经退出了这家公司, 仍然保留你所做设计或代码的副本往往是非法的。

## 5.2 如何设计可重用的代码

可重用的代码要满足两个主要目标。首先, 它要有足够的一般性, 可以用于稍有不同的多种用途, 或者可以在不同的应用领域中使用。如果程序组件带有特定应用的有关细节内容, 将很难在其他程序中重用。

可重用的代码还要易于使用。不需要花太多时间来理解其接口或功能。程序员必须能够很轻松地将其纳入到他们的应用中。

可重用代码要有一般性, 而且要易于使用。

你提供的可重用代码集并不一定非得是正式的库。它可以是一个类、一个函数集, 或者是一个程序子系统。不过, 与第4章一样, 这一章也使用“库”这个词泛指你编写的所有代码集。

需要注意的是, 本章使用“客户”一词表示使用你的接口的程序员。不要把这里的客户与运行你的程序的“用户”混为一谈。本章还使用“客户代码”一词表示为使用你的接口而编写的代码。

设计可重用代码时, 最重要的策略就是抽象。第2章通过真实世界中的电视对抽象做了对照分析, 你可以通过电视提供的接口来使用电视, 而无需了解其内部工作原理。类似地, 在设计代码时, 应当将接口与实现清楚地分离。这种分离使得代码更易于使用, 这主要是因为客户要使用此功能时, 不必了解其内容的实现细节。

基于抽象, 代码会分离为接口和实现, 所以设计可重用代码时所强调的主要就是这两个领域。首先, 必须适当地建立代码结构。你要使用哪些类层次体系? 是否应当使用模板? 应当怎样把代码划分为子系统?

其次, 必须设计接口 (interface), 这是库或代码的“入口”, 程序员就是使用这样一些入口来访问你提供的功能。要注意, 接口不一定是一个正式的 API。这个概念涵盖了你提供的代码与使用它的其他代码之间的任何边界。类的公共方法、函数原型的一个头文件等等就是很典型的接口。

“接口”一词可以表示一个访问点, 如一个单个的函数或方法调用, 或者可以表示一个完整的接口集, 如一个 API、类声明或头文件。

### 5.2.1 使用抽象

第2章已经学习了抽象的原则, 在第3章中对抽象在面向对象设计中的应用有了更多了解。要遵循

抽象原则，应当为代码提供一些接口来隐藏底层的实现细节。在接口和实现之间应当有一个清晰的界线。

使用抽象不仅对你有好处，对使用你代码的客户也有好处。客户会受益，这是因为他们不必操心实现细节就可以充分利用你提供的功能，而不必理解这些代码具体是如何工作的。你会受益，原因在于你能修改代码而不用修改代码的接口。因此，无需客户修改其使用代码，你也能提供升级和做出修正。利用动态链接库，客户甚至不需要重新生成其可执行程序。最后一点，你们都会受益，这是因为作为一个编写库的程序员，可以根据你希望提供什么样的交互以及想要支持什么样的功能来指定接口。接口和实现之间如果能清楚地分离，就可以避免客户采用你不期望的方式不适当地使用库，如若不然，这可能会导致预想不到的行为和 bug。

假设你在编写一个随机数库，而且希望为用户提供某种方法来指定随机数的范围。一种不好的设计是设置公共的全局变量或类成员，以供随机数生成器实现在内部使用，从而影响（改变）范围。这种设计得不好的库要求客户代码直接设置这些变量。好的设计则会隐藏内部实现所用的变量，而提供一个函数或方法调用来设置范围。如此一来，客户就不必理解内部算法。另外，由于实现细节未公开，你就能修改算法而不会影响客户代码与库的交互。

有时，库要求客户代码保留从一个接口返回的信息，以便将这些信息传递至另一个接口。这种信息有时称为一个句柄（handle），通常用于跟踪一些特殊的实例，这些实例需要记住调用之间的状态。如果你的库设计需要一个句柄，就不要暴露它的内部细节。要将该句柄置于一个不透明（opaque）的类中，程序员无法访问此类的内部数据成员。不要让客户代码调整这个句柄内部的变量。作为一个不佳设计的例子，本书作者之一就确实曾经用过一个不好的库，这个库要求他在一个原以为不透明的句柄中设置一个结构的特定成员，如此才能打开错误日志记录。

在编写类时，C++ 没有提供相关机制来支持好的抽象。必须将私有数据成员和方法声明放在公共方法声明所在的同一个头文件中。第 9 章介绍了一些技术，可以用这些技术绕过这些限制来提供清晰的接口。

抽象如此重要，它应当指导你的整个设计。在做每一个决策时，要问问自己你的选择是否满足抽象的原则。要从客户的角度来考量，并确定是否需要了解接口中的内部实现。一般不要违背这个原则。

### 5.2.2 适当地建立代码结构以优化重用

必须从设计的一开始就考虑重用。以下策略有助于你适当地组织代码。需要注意，这里所有策略都强调了代码的一般性。设计可重用代码的第二个方面是要易于使用，这与接口设计关系更大，将在本章后面介绍。

#### 避免将无关或逻辑上分离的概念混杂在一起

在设计一个库或框架时，要使它集中于一个任务或一组任务。不要将无关的概念（如随机数生成器和 XML 解析器）混杂在一起。

即使你并非专门设计重用的代码，也要牢记这个策略。很少会完全地重用整个程序。与此不同，程序中的部分或子系统会直接结合到其他应用中去，或者有所调整以适应稍有不同的用途。因此，应当适当地设计程序，从而将逻辑上分离的功能划分到不同的组件，这些组件可以在不同的程序中重用。

相对于真实世界中可互换的离散部件，上述编程策略正是对此设计原则的建模。例如，可以把一台旧车上的轮胎卸下来，换到另一种新车上使用。轮胎就是可以分离的组件，没有与车的其他部件固定在一起。卸轮胎时不用把发动机也一并拆下来！

在程序设计中，可以在宏观的子系统级和微观的类层次体系两方面应用这种逻辑划分策略。

### 把程序划分为逻辑子系统

要将子系统设计为可以独立重用的离散组件。例如，如果你在设计一个网络游戏，就应该把网络部分和图形用户界面部分放在不同的子系统中。这样一来，就可以分别重用这两个组件而不必在使用一个组件时牵扯上另一个组件。例如，你可能想编写一个非网络游戏，在这种情况下，可以重用图形用户界面子系统，而不需要网络部分。类似地，可以设计一个端到端（对等）文件共享程序，此时可以重用网络子系统，而不需要图形用户界面功能。

要确保每个子系统都遵循抽象原则，将子系统的接口与其底层实现清晰地分离。可以把每个子系统认为是一个小型库，必须为之提供一个一致而且易于使用的接口。即使只有你一个人使用这些小型的库，如果接口与实现设计得当，能将逻辑上不同的功能加以分离，也能获得很大好处。

### 使用类层次体系来分离逻辑概念

除了将程序划分为逻辑子系统，还要避免在类层次上将无关的概念混杂在一起。例如，假设你想为一个多线程程序编写一个平衡二叉树结构。你认为这个平衡二叉树数据结构一次应当只允许一个线程来存取或修改结构，因此，你在数据结构本身当中结合了加锁机制。不过，如果想在另外一些程序中使用这个二叉树的话，而这些程序刚好只是单线程程序，又会怎么样呢？在这种情况下，加锁只会浪费时间，而且要求程序必须与库链接，对于单线程程序，这本来是可以避免的。而且更糟糕的是，树结构可能在另一个平台上不能编译，因为加锁代码可能不是跨平台的。对此的解决方案就是创建一个类层次体系（见第3章的介绍），其中有一个线程安全的二叉树作为通用二叉树的一个子类。这样一来，就可以在单线程程序中使用二叉树超类，而不会不必要地引入加锁的开销，或者在一个不同的平台上使用二叉树超类，而无需重新编写加锁代码。图5-1显示了这个类层次体系。

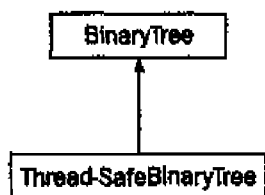


图 5-1

如果存在两个逻辑概念，如线程安全和二叉树，这种策略能很好地作用。如果有三个或更多概念，就会变得更为复杂。例如，假设你想同时提供一个  $n$  叉树和一个二叉树，它们可以是线程安全的，也可以不是。逻辑上讲，二叉树是  $n$  叉树的一个特例，因此应当是  $n$  叉树的一个子类。类似地，线程安全结构应当是非线程安全结构的子类。通过一个线性体系无法提供这些逻辑概念分离。一种可能的做法是让线程安全方面作为一个混合类，如图5-2中所示。

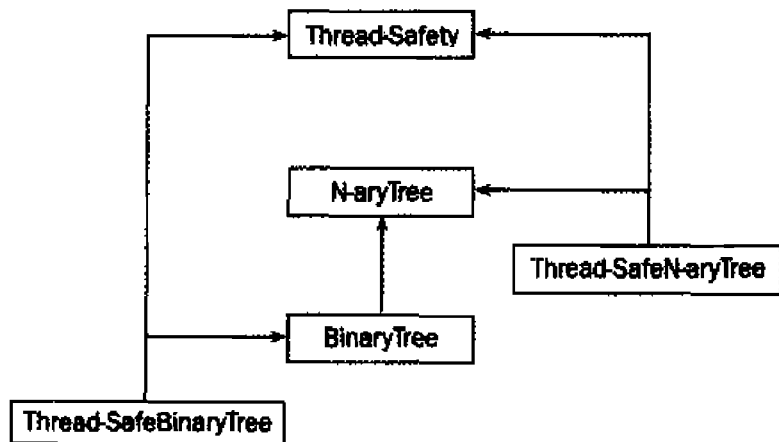


图 5-2

在这个层次体系中，尽管需要编写 5 个不同的类，不过由于能把功能清楚地分离，所以这是值得的。

还可以使用类层次体系将通用功能与更为特定的功能相分离。例如，假设你在设计一个操作系统，它支持用户级多线程。你可能想要编写一个包括多线程支持的进程类。不过，如果这些用户进程不想成为多线程的，怎么办呢？一个更好的设计是，创建通用进程类，并建立多线程进程作为它的子类。

### 使用聚集来分离逻辑概念

聚集 (Aggregation) 在第 3 章曾做过介绍，这是对 *has-a* 关系建立的模型，对象包含其他对象来完成其功能的某些方面。在继承不适用时，可以使用聚集来分离无关的功能，或者尽管相关但不同的功能。

还是看前面的操作系统例子，你可能希望将准备就绪进程存储在一个优先队列中，不要把优先队列结构与 ReadyQueue 类相集成，而应当编写一个单独的优先队列类。这样 ReadyQueue 类就可以包含并使用一个优先队列。按面向对象的术语来说，就是 ReadyQueue 有一个优先队列 (*has-a*)。利用这种技术，优先队列可以更容易地在另一个程序中重用。

### 对通用数据结构和算法使用模板

只要可能，就应当为数据结构和算法使用一种通用设计，而不是将特定程序的特定方面编写到代码中。不要编写一个只存储“书”对象的平衡二叉树结构。要让它作为一个通用结构，可以存储任何类型的对象。如此一来，你不仅可以将其用于一个书店、一个音乐商店、一个操作系统，而且只要是需要平衡二叉树结构的地方都可以使用。这种策略正是第 4 章所讨论的标准模板库 (STL) 的基础。STL 提供了可用于任何类型的通用数据结构和算法。

由 STL 可见，C++ 为这种通用编程模板提供了一个绝好的语言特性。如第 2 章和第 4 章所述，利用模板，就能编写用于任何类型的数据结构和算法。第 11 章提供了利用模板编写代码的详细内容，不过，这一节只讨论其中一些重要的设计方面。

### 为什么模板优于其他通用编程技术

模板并非唯一一种编写通用数据结构的机制。在 C 和 C++ 中也可以存储 `void*` 指针而不是一个特定类型，并以这种方式来编写通用结构。客户只需将某个类型强制转换为一个 `void*`，就可以使用这种结构来存储他们需要的任何东西。不过，这种方法存在一个主要问题，即它不是类型安全的，容器无法检查或保证所存储元素的类型。你可以将任何类型强制转换为一个 `void*`，从而存储在此结构中，而且从这个数据结构中删除指针时，必须再把它们强制转换回你所认为的类型。由于在此不涉及任何检查，其结果可能是灾难性的。可以假想有这样一种情况，一个程序员在一个数据结构中存储了指向 `int` 的指针，并且首先将其强制转换为 `void*`，但是另一个程序员可能认为这是 `Process` 对象的指针。第二个程序员会很自然地把 `void*` 指针强制转换为 `Process*` 指针，想把它们用作为 `Process*`。不用多说，程序肯定不能像他们预想的那样工作。

还有一种方法，即为特定类编写数据结构。通过多态，该类的任何子类都可以存储在这个结构中。Java 就把这种方法用到了极致，它指定每个类都直接或间接派生自 `Object` 类。Java 容器存储 `Object`，因此这些容器可以存储任何类型的对象。不过，这种方法不是真正类型安全的。从容器删除一个对象时，必须记住它实际上是什么，还要向下强制转换为适当的类型。

与此不同，模板如果使用得当则是类型安全的。模板的每个实例化只有一个类型。如果试图在同一个模板实例化中存储不同类型，程序就无法通过编译。

### 模板存在的问题

模板也不是十全十美的。首先，模板的语法很复杂，特别是以前从没有用过模板的人很可能摸不着头脑。其次，解析很困难，并非所有编译器都完全支持 C++ 标准。另外，模板要求同构的数据结构，在一个结构中只能存储相同类型的对象。也就是说，如果你编写了一个模板化的平衡二叉树，可以创建一

一个树对象来存储 Process 对象，再创建另一个树对象存储 int。但是不能在同一个树中同时存储 int 和 Process。这种限制是模板类型安全性的一个直接结果。尽管类型安全很重要，但是有些程序员认为这种同构需求是一个严重的限制。

模板存在的另一个问题是，它们会导致代码膨胀 (code bloat)。在创建一个树对象存储 int 时，编译器实际上会“扩展”模板来生成代码，就好像只为 int 编写了一个树结构一样。类似地，如果创建一个树对象来存储 Process，编译器也会生成代码，就好像只为 Process 编写了一个树结构。如果为多种不同类型实例化了模板，最后就会得到超大的可执行文件，因为在此会对应所有类型生成不同的代码。

### 模板 vs 继承

程序员有时会发现很难确定究竟该使用模板还是继承。下面是一个技巧，可以帮助你做出决定。

如果想为不同类型提供相同的功能，就可以使用模板。例如，如果想编写一个通用的排序算法，从而可以用在任何类型上，就应使用模板。如果想创建一个可以存储任何类型的容器，也可以使用模板。关键在于，模板化的结构或算法会对所有类型同等对待。

如果想为相关类型提供不同的行为，则应使用继承。例如，如果想提供两个不同但相似的容器，如队列和优先队列，此时就可以使用继承。需要注意，继承和模板可以结合使用。你能编写一个模板化的队列存储任何类型，而且有一个模板化优先队列作为其子类。第 11 章将介绍模板语法的有关详细内容。

### 提供适当的检查和防护

一定要将程序设计得尽可能地安全，以便其他程序员使用。这个原则最重要的方面就是要在代码中完成错误检查。例如，如果随机数生成器需要一个非负整数作为种子，不要寄希望于用户一定会正确地传入一个非负的整数。要检查传入的值，如果是非法值则应拒绝此调用。

作为对照，可以考虑一个帮你准备收入税费申报表的会计。当你雇用一位会计时，会向他或她提供你全年的财务信息。会计会使用这个信息来填写国家税务局 (Internal Revenue Service) 的表格。不过，这个会计并不会盲目地在表格中填入你的信息，而是会确保这些信息是有意义的。例如，如果你拥有一座房子，但是忘记指出你所交的财产税，会计就会提醒你补充提供这个信息。类似地，如果你说付的贷款利息是 \$12000，但总收入只不过 \$15000，会计可能会委婉地提醒你是不是数字给错了（或者至少建议你找一个更住得起的房子）。

可以把会计看作是一个“程序”，输入就是你的财务信息，输出是收入税费申报表。不过，会计的作用不只是会填写表格。你之所以会请一位会计，还因为他或她能够提供检查和防护。在编程中也是如此，类似地，应当在实现中提供尽可能多的检查和防护。

有许多技术和语言特性可以帮助你引入检查和防护。首先，可以使用异常来通知客户代码出现了错误。第 15 章将详细介绍异常。其次，可以使用智能指针（见第 4 章的讨论）和其他安全的内存技术（见第 13 章的讨论）。

### 5.2.3 设计可用的接口

除了抽象和适当地建立代码结构，要完成重用设计，还需要重视与程序员交互的接口。如果把一个丑陋的实现隐藏在一个很漂亮的接口里，没有人会知道。不过，如果你提供了一个很不错的实现，但其外部的接口很糟糕，那么你的库不会好到哪去。

需要注意，程序中每个子系统和类都应当有好的接口，即使你本来不打算在多个程序中使用这些子系统和类也不例外。首先，你无法知道什么时候可能会得到重用。其次，即使对于第一次使用，好的接口也很重要，特别是当你在一个团队中参与开发，而其他程序员必须使用你设计和编写的代码，好接口则更为重要。

接口的主要用途就是使代码易于使用，不过有些接口技术还可以帮助你遵循一般性原则。



### 设计易于使用的接口

你的接口应当易于使用。这并不是说这些接口要很平常，而是说在满足功能的条件下要尽可能地简单和直观。如果使用你的库的人要通读数页的源代码才能使用一个简单的数据结构，这就很不好，或者，要得到所需的功能，他们必须对代码做大幅调整，这也是应该避免的。本节对于如何设计易于使用的接口提供了 4 个具体的策略。

#### 开发直观的接口

计算机程序员用直观 (intuitive) 一词来表示这种接口很容易领会，而无须太多讲解。“直观”这个词和短语“显而易见”在含义上是类似的，后者表示不用太多解释或分析就很明显。从定义几乎可以了解到，使用直观的接口会很容易。

要开发直观的接口，最好的策略就是遵循标准做法和人们最熟悉的做法。当人们遇到一个接口，而这个接口与他们以前使用过的某个接口很像，就能更好地理解、更容易地采纳，而且一般不太会使用不当。

例如，假设你在开发一辆车的操纵机制。对此可以有多种可能，可以是一个操纵杆、两个左移或右移的按钮、一个滑动水平杆或者是一个常规的方向盘。你觉得哪一个最容易使用呢？觉得用哪一个接口的车最好卖呢？顾客一般对方向盘很熟悉，所以这两个问题的答案自然都是方向盘。即使你开发过另外一种机制可以提供更好的性能和安全性，但是卖这种产品的时候你肯定会遇到困难，更不用说教人们学习如何使用这种产品了，可以想见难度会有多大。到底是遵循标准接口模型，还是另辟蹊径呢？在这二者之间做出选择时，最好还是采用人们熟悉的接口。

创新固然重要，但是应该强调底层实现中的创建，而不是接口中的创建。例如，顾客非常青睐某些车模里混合式的油电双用发动机。这些车在这个方面就卖得很好，因为这些车的接口与使用其他标准发动机的车没有什么不同。

如果应用到 C++，这种策略则说明你开发的接口应当遵循 C++ 程序员熟悉的标准。例如，C++ 程序员希望类的构造函数和析构函数会分别初始化和清除对象。在设计类时，就应当遵循这个标准。如果你要求程序员自行调用 `initialize()` 和 `cleanup()` 方法来完成初始化和清除，而不是把这些功能放在构造函数和析构函数中，使用这个类的每一个人都会搞糊涂。因为你的类与其他 C++ 类表现不同，程序员学习如何使用这个类就要花费更长的时间，而且忘记调用 `initialize()` 或 `cleanup()` 的可能性更大（以至于使用不正确）。

一定要从使用接口的人的角度来考虑接口。它们有意义吗？这正是你所期望的吗？

C++ 提供了一种称为操作符重载 (operator overloading) 的语言特性，这有助于你为对象开发直观的接口。利用操作符重载，可以编写类，从而使用标准操作符也可以处理这些类，就好像处理诸如 `int` 和 `double` 等内置类型一样。例如，可以编写一个 `Fraction` 类，从而完成加法、减法和流操作，如下所示：

```
Fraction f1(3,4), f2(1,2), sum, diff;
sum = f1 + f2;
diff = f1 - f2;
cout << f1 << " * " << f2 << endl;
```

与之对照，如果使用方法调用来完成同样的行为，则如下所示：

```
Fraction f1(3,4), f2(1,2), sum, diff;
sum = f1.add(f2);
diff = f1.subtract(f2);
f1.print(cout);
cout << " * ";
f2.print(cout);
cout << endl;
```

可以看到,利用操作符重载,可以为类提供直观的接口。不过,要注意不要滥用操作符重载。有可能会重载+操作符,使之实现减法;而把-操作符重载为实现加法。这些实现就与直观背道而驰。一定要遵循这些操作符本身的一般含义。有关操作符重载的详细内容请参见第9章和第16章。

#### 不要遗漏必要的功能

这个策略有两个方面。首先,要在接口中包括客户可能需要的所有行为。乍一看这可能很显然。还是对照着车来分析,你肯定不会造一辆没有速度表的车(否则驾驶员就无法了解他或她的速度是多少了!)。类似地,你也不会设计未提供访问机制的Fraction类(也就是说,不允许客户代码访问这个分数的实际值)。

不过,其他行为可能没有这么明显。这种策略要求你预计到代码的所有用途,即客户会把你的代码用到哪里。如果以某个特定的方式来考虑接口,可能会遗漏一些功能,而客户以另一种方式使用代码时会需要这些功能。例如,假设你想设计一个游戏棋盘类。可能只考虑了典型的游戏,如象棋和跳棋,因此决定支持一种每一点上最多只有一个棋子的棋盘。不过,如果后来你又决定编写一个巴加门15子棋游戏,这样棋盘上每个点允许有多个棋子,又该怎么办呢?由于当初完全杜绝了这种可能性,就无法将前面写的棋盘用作为巴加门棋盘了。

显然,要想预计到库的每一种可能的用途,这往往很难,甚至是不可能的。不要有压力,不要因为想设计完美的接口而力图考虑到将来所有可能的用途,只要能充分考虑,并且尽你所能就可以了。

这个策略的第二个方面是在实现中尽可能多地加入功能。如果在实现中已经知道某些信息,或者另外采用适当的设计就可以从实现中知道有关信息,那就不要让客户代码来指定这些信息。例如,如果你的XML解析器库需要一个临时文件来存储结果,不要让使用这个库的客户来指定该文件的路径。这些客户并不关心你用的是什文件,应当另找办法来确定一个合适的文件路径。

另外,不应要求库用户完成不必要的工作来得到合并的结果。如果随机数库使用了一个随机数算法,而它要单独计算随机数的低位和高位,就应该将其低位数和高位数先行合并,然后再提供给用户。

如果你能为用户完成某些任务,就不要让库的用户自己来完成。

#### 提供简洁的接口

为了避免在接口中漏掉功能,有些程序员会走入另一个极端,他们会在接口中包括可能想到的每一个功能,并认为这样一来,使用接口的程序员肯定能找到完成任务的办法。遗憾的是,这种接口可能太过杂乱,以至于程序员根本无法了解该怎么做!

不要在接口中包括不必要的功能,要让接口保持清楚而简单。最初看来,这个原则与上一条策略(不要遗漏必要的功能)好像是矛盾的。为了避免遗漏功能,可以在接口中包括每一个可能想到的功能,尽管这也是一个策略,但是这不是一个健全的策略。正确的策略是:应该包括必要的功能,而去掉无用或无意义的接口。

可以再来考虑汽车的例子。开车时只会与几个组件打交道:方向盘、刹车和油门、变速器、后视镜和仪表盘上的其他一些仪表。下面假设一个汽车的仪表盘就像飞机的仪表盘一样,有数百个仪表、操纵杆、监控器和按钮。这就很没用。开车可比开飞机容易多了,所以接口可以简单得多,你不用查看你的高度,不用与控制塔通信,也不用控制飞机上的许多组件,如机翼、引擎和着陆装置等等。

另外,从库开发的角度看,小一些的库也更易于维护。如果你想让每一个人都满意,这往往会导致出错的可能性更大,如果实现太过复杂,所有一切都交织在一起,那么即使只有一个错误,也可能导致库无法使用。

遗憾的是,要设计简洁的接口,这种想法从理论上看起来很好,但是要付诸实践却极为困难。这个原则

往往存在主观性，你要确定哪些是必要的，哪些是不必要的。当然，如果你的选择有错，客户肯定会指出来。以下是务必记住的一些技巧。

- 消除重复的接口。如果一个方法以英尺为单位返回一个结果，另一个方法以米为单位返回一个结果，可以把这二者合并为一个方法，返回一个对象，从而既可以按英尺又可以按米提供结果。
- 确定采用何种方法提供所需的功能最为简单。去掉不必要的参数和方法，并适当地将多个方法合并为一个方法。例如，如果有一个方法可设置用户指定的初始化参数，就可以把它与一个库初始化例程合并。
- 适当地限制库的使用。要想满足每个人的要求，这是不可能的。不可避免，肯定有人会以某种你不希望的方式来使用库。例如，如果你提供了一个库来完成 XML 解析，有人可能会用它来解析 SGML。这并非你的本意，所以对此不必有压力，不需要考虑还对 SGML 解析提供支持。

### 提供文档和注释

不论你的接口多么容易使用，多么直观，都应当提供相关文档来说明这个接口如何使用。除非你告诉了其他程序员该如何使用你写的库，否则不要寄希望他们总会正确地使用这个库。可以把你的库或代码认为是其他程序员消费的产品。你购买的所有有形产品（如 DVD 播放器）都会提供一组说明来解释其接口、功能、限制和有关问题。即使是椅子之类简单的产品通常也提供了说明，来指出如何正确地使用，尽管这个说明可能相当简单，也许只是“可以坐在本产品上。如果用其他方式使用本产品，可能带来人身伤亡”。但这确实是对产品功能的一个说明。类似地，你的产品也应该提供文档来解释如何正确地使用。

为接口提供文档有两种方法：在接口本身提供注释，以及外部文档。应当这两方面都做到才好。大多数公共 API 只提供了外部文档，许多标准 UNIX 和 Windows 头文件中很少有注释。在 UNIX 中，通常以在线手册的形式提供文档，这些在线手册称为手册页（man page）。在 Windows 中，集成开发环境会随附文档。

尽管大多数 API 和库在接口本身中都没有（或很少有）注释，但我们确实认为这种文档才最为重要。不应该发布一个只包含代码而没有任何注释的“裸”头文件。即使注释与外部文档中的说明是重复的，与查看只有代码的头文件相比，如果这个头文件中有友好的注释，就不会那么让人心生畏惧了。即使是最好的程序员，还是希望经常能看到文字性的说明。

有些程序员会使用一些工具从注释自动创建文档。这些工具会解析带有特殊关键字和格式的注释来生成文档，通常最终的文档采用超文本标记语言（Hypertext Markup Language, HTML）格式。Java 程序设计语言提供了 JavaDoc 工具，使得这个技术进一步得到推广，不过对于 C++ 也有许多类似的可用工具。第 7 章将更为详细地讨论这个技术。

在提供注释、外部文档或者二者都提供时，文档应当描述库的行为，而不是实现。行为包括输入、输出、错误条件和处理、本来用途和性能保证。例如，对于一个生成随机数的调用，描述此调用的文档应当指出它无需任何参数，返回一个指定范围内的整数，如果无法分配内存则抛出一个“内存不够”异常。这种文档不应解释具体生成随机数的线性算法细节。这个接口的客户并不关心算法，只要生成的数看上去是随机的就行！

在接口注释中提供太多实现细节可能是接口开发中最常见的一个错误。我们见过许多这样的例子，本来接口和实现得到了很好的分离，但却因为接口中的注释搞砸了，这些注释不应提供给客户，而应提供给库的维护人员才更合适。

公共的文档应当指定行为，而不是底层实现。

当然对内部实现建立文档是必要的，但不要把它作为接口的一部分公开。对于如何在代码中适当地使用注释，第7章提供了更详细的说明。

#### 设计通用接口

接口应当足够的通用，以便用于各种不同的任务。如果在一个原本打算通用的接口中写入了某个应用的具体细节，它就无法用于任何其他用途了。以下是一些要记住的原则。

#### 提供多种方法来完成同一功能

为了满足所有“顾客”的要求，有时提供多种方法来完成同一功能会很有帮助。不过，要明智地使用这个技术，因为过分使用很容易导致杂乱的接口。

再来考虑车的例子。如今大多数新车都提供了不用钥匙的远程开锁系统，只用按下遥控钥匙上的一个按钮就可以开锁。不过，这些车还是会提供一个标准的钥匙，可以用这个钥匙真正地锁车。尽管这两种方法是冗余的，但大多数顾客还是很高兴有这样两种选择。

在程序接口设计中，有时也有类似的情况。例如，假设某个方法要取一个字符串作为参数。可能希望提供两个接口：一个取 C++ string 对象，另一个取 C 风格的字符指针。尽管这二者之间可以转换，但是不同的程序员可能倾向于使用不同类型的字符串，所以同时提供这两种方法很有帮助。

需要注意，这种策略可以认为是接口设计中“简洁”原则的一个例外。在很多情况下这种例外是合适的，不过还是要更多地遵循“简洁”原则。

#### 提供定制能力

为了提高接口的灵活性，需要提供定制能力。在没有定制能力的时候，人们对此往往有迫切的需要。例如，本书作者之一最近购买了一辆带防盗器的新车。如果用远程无钥开锁系统开锁，警报会自动解除。遗憾的是，如果开锁后 30 秒内车门没有打开，警报又会启动。在往车的后备箱放东西或拿东西时，有时这个特性很是烦人。如果只是要拿后备箱里的东西，打开车门会很不方便。不过，如果没有解除警报，关后备箱盖时又会让警报哗哗作响。这个问题最恼人的地方是，没有办法永久地解除防盗器。车的设计者肯定认为每个人都希望他们的防盗器有相同的功能，因此没有提供可以定制的机制。

定制能力可能很简单，只是允许客户打开或关掉错误日志记录。定制能力的基本前提是，可以为每个客户提供同样的基本功能，但是允许客户稍微调整。

通过函数指针和模板参数，可以提供更大的定制能力。例如，可以让客户设置自己的错误处理例程。这种技术就是装饰器模式（见第 26 章的介绍）的一个应用。

STL 将这种定制能力策略用到极致，允许客户为容器指定自己的内存分配器。如果想使用这个特性，就必须编写一个遵循 STL 原则的内存分配器对象，并满足所需的接口。STL 中的每个容器都取一个分配器作为它的一个模板参数。第 23 章将介绍有关的详细内容。

### 5.2.4 协调一般性和易用性

一方面想要易于使用，另一方面又想做到一般性，这两个目标有时好像存在着冲突。通常，引入一般性会增加接口的复杂性。例如，假设在一个地图程序中需要一个图结构来存储城市。为了做到一般性，可能会使用模板来编写一个适用于任何类型的通用地图结构，而不只是存储城市。这样一来，如果想在下一个程序中编写一个网络仿真器，就可以采用同一个图结构来存储网络中的路由器。遗憾的是，使用模板的话，接口可能不太漂亮，而且较难使用，特别是当客户不熟悉模板时更是麻烦。

不过，一般性和易用性并不是完全互斥的。尽管在某些情况下一般性的增加会降低易用性，但是设计出既一般又易于使用的接口并非没有可能。可以遵循以下两个原则。

#### 提供多个接口

为了减少接口的复杂性，但仍然提供足够的功能，可以提供两个单独的接口。例如，可以编写一个通用的

网络库，其中包括两个单独的部分。其中一部分提供对游戏有用的网络接口，另一部分则提供对超文本传输协议（Hypertext Transport Protocol, HTTP）Web 浏览协议有用的网络接口。

STL 对其 `string` 类就采用了这个方法。正如第 4 章所提到的，`string` 类实际上就是 `basic_string` 模板的一个 `char` 实例化。可以把类认为是一个接口，它隐藏了整个 `basic_string` 模板的复杂性。

### 优化常用功能

提供一个通用接口时，有些功能可能比其他一些功能用得更多。应当让常用的功能易于使用，而仍然提供更高级的功能以便选择。再来看地图程序，你可能希望为地图客户提供一个选项，允许用户使用不同的语言指定城市的名字。由于英语如此普及，因此可以将英语作为默认语言，但是还提供一个额外的选项来改变语言。这样一来，大多数客户都不必担心语言的设置问题（也就是说，可以采用默认的英语），但是如果客户确实想采用其他语言，也可以达到目的。第 4 章中曾谈到要优化最常执行的部分代码，上述策略与第 4 章讨论的这个性能原则很接近。通过重点强调对设计中这些最常用部分进行优化，就能让大多数人得到最大好处。

## 5.3 小结

通过阅读本章，你学习了为什么要编写可重用的代码，以及如何编写可重用的代码。在此不仅了解了重用方法论（可以总结为“编写一次，经常使用”），并且知道了可重用代码应当既具有一般性，而且要易于使用。你还会发现，设计可重用代码有三个要求：使用抽象、适当地建立代码结构，而且要设计好的接口。

关于如何建立代码结构，本章给出了三个具体的技巧：要避免将无关或逻辑上分离的概念混在一起、使用模板建立通用数据结构和算法、提供适当的检查和防护。

本章还提供了 6 个设计接口的策略：开发直观的接口、不要遗漏必要的功能、提供简洁的接口、提供文档和注释、提供多种方法来完成同一功能、提供定制能力。最后对如何协调两个经常冲突的需求（一般性和易用性）给出了两点提示：提供多个接口、优化常用功能。

到本章为止，从第 2 章开始介绍的设计主题就结束了。第 6 章将讨论软件工程方法论，并以此结束本书第一部分有关设计的内容。第 7 章～第 11 章将深入到软件工程过程的实现阶段，并介绍具体的 C++ 代码编写。

## 第6章 充分利用软件工程方法

刚开始学习如何编写程序时，你可能有自己的安排。也许随意性很大，把工作放在最后一刻才做，也可能在实现过程中完全改变主意。不过，如果是专业性地编写代码，程序员很少会有这种灵活性。即使是思想最活跃的工程管理人员也承认，一定的过程管理很有必要。如今，了解软件工程过程与了解如何编写代码同样重要。

这一章会分析多种软件工程方法。在此不会对任何一种方法做非常深入的介绍，有关软件工程过程已经有许多非常好的书了。我们的想法是拓宽思路，介绍多种不同类型的过程，使你能够有所比较、有所对照。我们不会说哪一种方法特别好，也不会说哪一种方法特别不好。相反，希望通过不同方法之间的权衡，自己做出决定，建立一个适合自己 and 小组中其他成员的软件工程过程。

不论你是独立地开发项目，还是参与一个由来自各大洲数百位工程人员组成的团队，如果能理解软件开发的不同方法，将有助于日常的工作。

### 6.1 为什么需要过程

在软件开发的历史进程中，不乏项目惨败的事件发生。从预算超支且销售业绩很差的客户应用，到超大规模的操作系统，几乎没有哪个软件开发领域能逃过此劫。

即使软件确实到了用户手中，但由于 bug 太多，以至于最终用户不得不忍受不断的更新和补丁。有时软件并没有完成它本应完成的任务，或者并没有按用户期望的方式来完成。由这些问题可以清楚地指出软件界众所周知的一点：编写软件确实很难。

有人可能会感到奇怪，同样是工程，与其他形式的工程相比，为什么软件工程会这么频繁地失败呢？虽然汽车也存在问题，但你很少看到汽车会因为缓冲区溢出突然停下来，而要求重新启动（不过，随着越来越多的汽车组件都由软件驱动，这种情况也是有可能的！）。举例来说，电视可能并非完美无缺，但是起码不必升级到 2.3 版本才能收到 6 频道。

这是不是说其他工程领域比软件工程更高级呢？城建工程人员能够从修建大桥的源远流长中汲取经验来建造一座合用的大桥。化学工程人员能够成功地合成一种化合物，因为已经通过前几代的试验解决了其中存在的问题。是不是这样呢？

软件（领域）是不是太新了，或者是不是它本身存在一些特性确实会导致 bug 的出现、无法使用的结果，以及招致项目失败？

看上去似乎与软件确实有所不同。一方面，在软件中，技术会迅速改变，这就带来软件开发过程的不确定性。即使在开发项目过程中没有遇到地震量级的突破，业界飞快前进的步伐还是会带来巨大的冲击。通常需要快速地开发软件，因为竞争太过激烈了。

软件开发也可能是不可预料的。要想做出精确的进度安排几乎是不可能的，因为仅仅一处小 bug 就可能需要花费数天甚至几个星期才能解决。即使一切似乎都按进度进行，但产品定义（产品需求）极有

可能改变,这就是功能扩张(feature creep),而这会严重影响甚至改变进程。

软件很复杂。没有一种容易而且准确的方法能够证明一个程序没有 bug。如果软件经过维护,发展了多个版本,倘若代码存在 bug 或者很混乱,就会对软件造成长时间(几年)的影响。软件系统通常非常复杂,这样当有人员调动时,对于粗心大意的人所留下的混乱代码,没有人乐意去靠近(接手)。而复杂的软件会导致一个无休止的循环往复:打补丁、修改,再解决问题。

当然,软件中还存在一般的企业风险。市场压力和交流不畅仍然存在。许多程序员想要理清公司的行政事务,但是由于市场压力的存在和交流不畅,开发小组与产品营销小组之间可能会出现争议,这种情况绝非少见。

软件工程产品所存在的这些负面因素都说明需要某种过程管理。软件项目规模很大、很复杂,而且前进的步伐很快。为了避免失败,工程小组需要采用一个系统来控制这种难处理的过程。

## 6.2 软件生命期模型

软件中的复杂性并不是一个新内容。早在数十年前人们就已经意识到形式化过程的必要性。已经提出了许多软件生命期(software life cycle)的建模方法,力图在软件开发的混乱局面中引入一些秩序,即按步骤来定义从最初提出软件想法到形成最终产品之间的软件过程。这些模型经过多年的改进,正指导着当前的许多软件开发。

### 6.2.1 分阶段模型和瀑布模型

软件的经典生命期模型一般称为分阶段模型或分步模型(Stagewise Model)。这个模型的基本思想是软件可以像照着菜谱一样来构建。相应地有一组步骤,如果严格地按照菜谱,就能得到一个很美味的巧克力蛋糕,类似地,如果确实按步骤进行,就能得到所期望的程序。下一阶段开始之前,前一个阶段必须先完成,如图 6-1 所示。

这个过程先从正式的规划开始,包括收集大量需求。这个需求表可以决定产品的功能是否完备。需求越具体,项目就越有可能成功。接下来,要设计并明确地指定软件。设计步骤与需求步骤类似,也要尽可能地具体,这样才更有可能成功。所有设计决策都在这一步做出,通常要提供伪代码,对于所需编写的特定子系统,还要给出相应的定义。编写子系统的人要明确其代码如何交互,开发小组要协商好体系结构的特定细节。接下来是对设计加以实现。由于已经明确地指定了设计,代码必须严格地遵循设计,否则各部分代码将无法协作。最后 4 个阶段则分别完成单元测试、子系统测试、集成测试和评估。

分阶段模型的主要问题在于,在实际中,一般至少需要对下一个阶段做些分析,否则在此之前完成前一个阶段几乎是不可能的。如果不写些代码,设计往往不能确定。另外,如果这个模型没有提供适当的途径返回编码阶段(即再对代码做些调整),那么测试还有什么意义呢?

对分阶段模型提出了许多改进,这些改进在 20 世纪 70 年代初提出的瀑布模型(Waterfall Model)中得到了形式化。这个模型仍然对当前的软件工程组织有很大的影响(甚至是主导性的影响)。瀑布模型所引入的主要改进是提出了阶段之间可以存在反馈。尽管它还是强调规划、设计、编码和测试这样一个严格的过程,

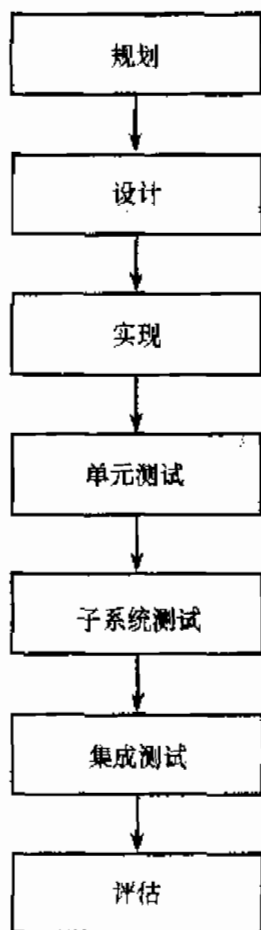


图 6-1



但是后面的阶段可以部分重叠。图 6-2 显示了瀑布模型的一个例子，在此可以看到反馈和重叠改进。基于反馈，在某个阶段获得的教训可以导致前一个阶段的修改。基于重叠，则允许两个阶段的行动可以同时发生。

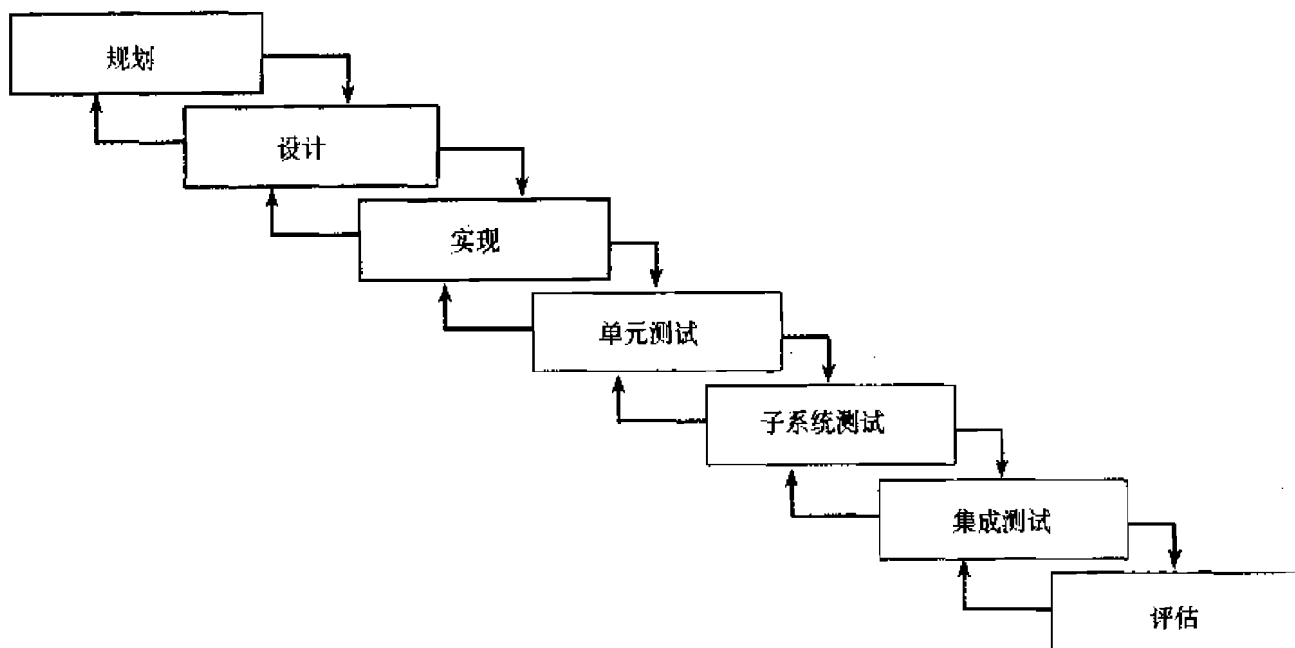


图 6-2

不同的瀑布方法会以不同的方式改进这个过程。例如，有些规划会包括一个“可行性”步骤，即在收集正式的需求之前，先完成一些可行性试验。

#### 瀑布模型的好处

瀑布模型的价值在于它相当简单。实际上，你或管理人员在以往的项目中可能已经采用过这种方法，只不过不是那么正式，或者没有用这个名字而已。分步（分阶段）模型和瀑布模型有一个前提假设：只要每一步尽可能完全、尽可能准确地完成，后面的步骤就可以顺利地进行。只要在这一步仔细地指定了所有需求，而且第 2 步中做出了所有设计决策，并解决了所有问题，那么第 3 步中的实现就只是把设计翻译为代码而已。

由于瀑布模型的简单性，这就使得基于这种系统建立的项目规划很有组织性，而且很容易管理。使用这种方法的管理人员可能会提出要求，例如在设计阶段的最后，负责各子系统的所有工程人员要将其设计作为一份正式的设计文档或功能子系统规范提交上来。对于管理人员来说，其好处在于，由工程人员先行指定并完成设计，就能尽可能地降低风险。

从工程人员的角度看，采用瀑布方法可以提前明确主要问题。所有工程人员在编写大量代码之前都需要了解项目，并设计其子系统。理想情况下，这意味着代码可以只编写一次，而不是等发现各部分代码不能很好地协作时，再把各部分代码纠缠在一起进行修改或完全重写。

对于有特定需求的小型项目，瀑布方法能很好地工作。特别是在顾问领域，它的优点在于可以从项目一开始就明确可以用哪些特定的度量标准来衡量项目成功与否。将需求形式化，这有助于顾问准确地得出客户希望做什么，并要求客户具体地指出项目的目标。

#### 瀑布模型的缺点

在许多组织中，以及在几乎所有软件工程方面的资料中，瀑布方法都不再得宠。瀑布方法认为软件

开发任务总在离散的线性步骤中完成，但有批评指出这个基本前提是不实际的。尽管瀑布方法允许阶段的重叠，但是并不允许较大幅度的后退。在当前的许多项目中，产品的整个开发过程中都可能出现新的需求。通常，顾客可能需要一个新特性，而且只有满足了这个特性，产品才卖得出去，或者是竞争对手的产品提供了一项新功能，所以你的产品中也需要有一个与之相当的功能。

提前指定所有需求，这使得瀑布方法对许多组织来说都是不可用的，因为这样就无法做到足够的灵活（动态）。

还有一个缺点是，为了尽可能降低风险，瀑布模型会尽可能正式而且尽早地做出决策，而这实际上可能只是把风险隐藏起来。例如，在设计阶段可能没有发现、遗漏、忘记或有意避开了一个主要的设计问题。到了集成测试阶段才发现存在这个问题，此时要想挽救这个项目可能已经为时过晚了。尽管已经暴露出了一个重大的设计缺陷，但是按照瀑布模型，并没有解决这个问题，而是带着它更进一步！瀑布过程中任何一个错误都很可能导致过程最后的失败。很难做早期检测，也很少这么做。

尽管瀑布模型仍然很常用，而且这仍是过程进行直观分析的一个有效手段，但是通常很有必要采纳其他方法的一些优点，让它更灵活一些。

### 6.2.2 螺旋方法

螺旋方法（Spiral Method）是 Barry W. Boehm 在 1988 年提出的，他认识到软件开发过程中可能出现未预料到的问题，而且需求可能发生改变，因此提出了这种方法。这个方法只是一组称为迭代过程（iterative process）技术中的一部分。其基本思想是，如果哪里出了问题，也没有关系，因为你会在下一轮里解决这个问题。图 6-3 显示了螺旋方法中的一轮（一次迭代）。

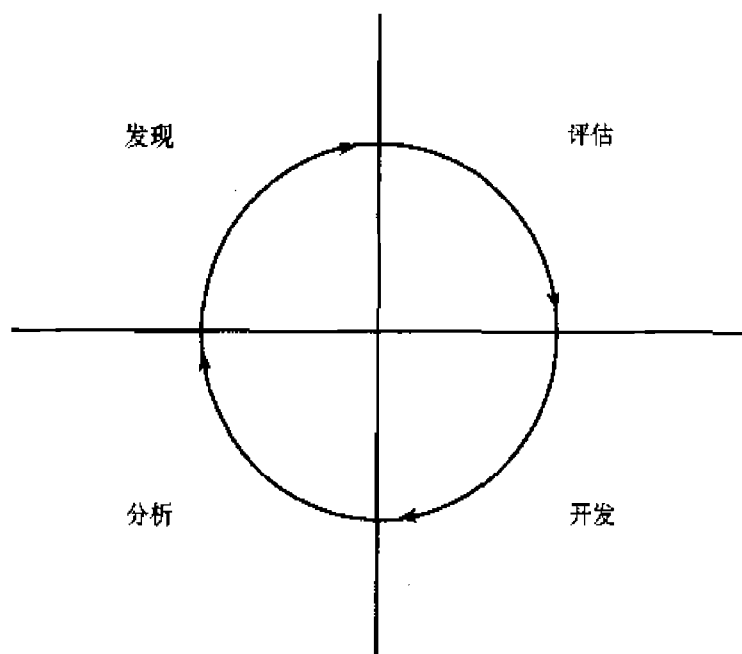


图 6-3

螺旋方法的各个阶段与瀑布方法中的各个步骤很相似。发现阶段包括建立需求，并确定目标。在评估阶段，会考虑多种实现选择，并且可能会建立原型。螺旋方法中，会特别重视评估阶段的风险评估和风险消除。螺旋的当前周期实现的是风险最大的任务。开发阶段的任务就由评估阶段明确的风险来确定。

例如，如果通过评估发现一个有问题的算法可能无法实现，那么当前周期的主要开发任务就是建立该算法模型、构建算法并测试。第4阶段用于完成分析和规划。基于当前周期的结果，可以建立后续周期的规划。每次迭代都要在较短的时间内完成，只考虑不多的几个关键特性和风险。

图6-4显示了采用螺旋方法开发一个操作系统的三个周期。第一个周期得到了一个包含产品主要需求的规划。第二个周期得到一个反映用户体验的原型。第三个周期建立一个组件，因为经认定这个任务的风险很高。

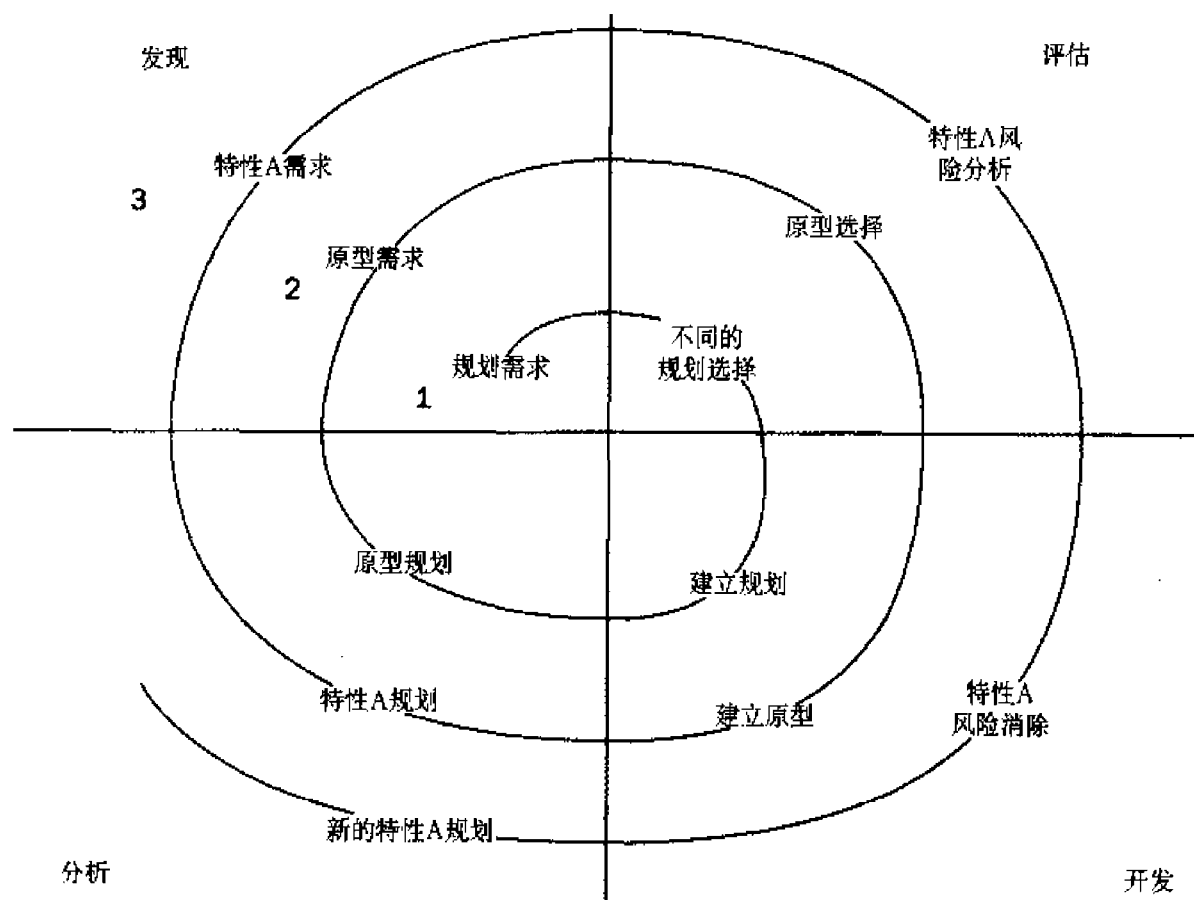


图 6-4

### 螺旋方法的好处

瀑布方法应当提供迭代，而螺旋方法可以看作是这种迭代方法的极致应用。图6-5将螺旋方法显示为一个瀑布过程，这个过程已经得到调整以允许迭代。通过这些为时不长的迭代周期，就可以消除瀑布方法存在的主要缺点（隐藏风险和线性开发路线）。

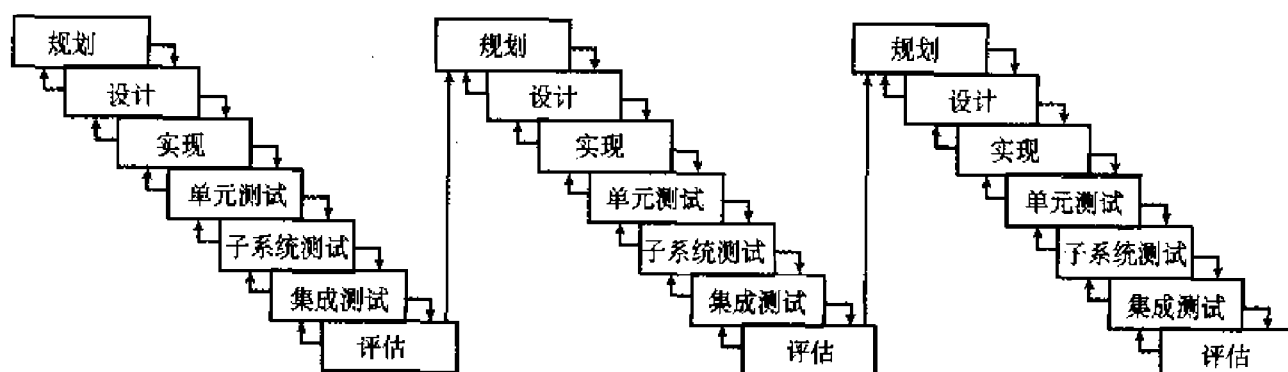


图 6-5

螺旋方法还有一个好处,即风险最大的任务最先完成。通过把风险放到前台,并认识到任何时刻都可能会有新的条件发生,螺旋方法就能避免瀑布模型中可能出现的隐藏的时间炸弹。出现未预料到的问题时,可以使用同样的4步方法来处理,这对余下的过程也同样适用。

最后一点,每个周期之后都进行分析并建立新的设计,通过这种重复做法,基本上可以消除“设计-然后实现”方法所存在的实际困难。通过每个周期,会对系统有更多了解,这些了解会进一步影响设计。

#### 螺旋方法的缺点

螺旋方法的主要缺点在于,要将每次迭代界定得足够小,以便得到真正的收益,这往往很困难。在最坏的情况下,螺旋方法会蜕化为瀑布模型,因为一次迭代可能太长了。遗憾的是,螺旋方法只是对软件生命期建模。它无法指定一种具体的方法将项目分解为单周期的迭代,原因是这种划分会因项目不同而有所不同。

除此以外,螺旋方法还存在其他一些可能的缺点,这包括每个周期都要重复这4个阶段,这就会带来一定的开销,另外协调各个循环周期存在着难度。从逻辑上讲,可能很难在适当的时间将小组成员对设计的有关讨论汇总在一起。如果不同的小组在同时完成产品的不同部分,就有可能在并行的循环周期中工作,而这可能是不同步的。例如,开发用户界面的小组可能准备启动窗口管理器(Window Manager)周期的发现阶段,而核心操作系统(OS)小组还处在内存子系统的开发阶段。

### 6.2.3 统一开发过程

统一开发过程(Rational Unified Process, RUP)是管理软件开发过程的一种严格的形式化方法。RUP最重要的特征在于,不同于螺旋方法或瀑布模型,RUP不仅仅是一种理论性过程模型。RUP实际上是一个软件产品,由Rational Software(这是IBM的一个部门)推向市场。完全可以把这个过程看作是软件,主要有以下几个原因:

- 过程本身可以得到更新和改进,就像软件产品会周期性地更新一样。
- RUP并不只是推荐了一个开发框架,它还包括有一组软件工具,可与该框架结合使用。
- 作为一个产品,RUP可以推广到整个工程小组,这样所有成员都会采用完全相同的过程和工具。
- 与许多软件产品一样,RUP可以定制,以满足用户的需要。

#### RUP 作为产品

作为一个产品,RUP的形式是一组软件应用,可在软件开发过程中为开发人员提供指导。RUP产品还可为其他Rational产品提供特殊的指导,如Rational Rose可视化建模工具和Rational ClearCase配置管理工具。另外RUP包括大量群件(groupware)通信工具,作为“思想市场”(marketplace of ideas)的一部分,开发人员可以利用这些工具实现知识共享。

RUP的一个基本原则是:开发周期的每次迭代都应当有一些有形的结果。在统一开发过程中,用户会建立许多设计、需求文档、报告和计划。RUP软件为建立这些结果提供了可视化工具和开发工具。

#### RUP 作为过程

定义一个准确的模型,这是RUP的核心原则。按照RUP的说法,模型有助于解释软件开发过程中复杂的结构和关系。在RUP中,模型通常用统一建模语言(Unified Modeling Language, UML)的格式来表示。

RUP将过程的每个部分定义为一个单独的工作流(workflow)。工作流从以下几个方面来表示过程中的每一步,包括谁负责这一步,要完成哪些任务,这些任务的结果是什么,以及驱动任务的事件序列等。RUP的所有一切都是可定制的,不过RUP“预先”定义了一些核心过程工作流(core process workflow)。

核心过程工作流与瀑布模型中的阶段有一定的相似性,但是每个核心过程工作流都是迭代的,而且在定义上更为特定。业务建模工作流(business modeling workflow)是对业务过程建模,其目标通常是驱动软件需求向前发展。需求工作流(requirements workflow)通过分析系统中的问题,并描述其假设,从而得到需求定义。分析和设计工作流(analysis and design workflow)所处理的是系统体系结构和子系统设计。实现工作流(implementation workflow)涵盖了软件子系统的建模、编码和集成。测试工作流(test workflow)是对软件质量测试的规划、实现和评估建模。部署工作流(deployment workflow)是对整体规划、发布、运行和测试工作流的一个高层视图。配置管理工作流(configuration management workflow)涉及从新的项目概念出发,通过反复迭代,最终得到产品的过程。最后,环境工作流(environment workflow)通过创建和维护开发工具为工程组织提供支持。

### RUP 实践

RUP 主要在较大规模的组织中采用,与传统的生命周期模型相比,RUP 有许多优点。一旦开发小组完成了学习曲线,了解了如何使用这个软件,所有成员都会使用一个共同的平台来设计、交流和实现其思想。这个过程可以进行定制,以满足小组的特定需求,而且每个阶段都能得到大量有价值的结果,这就是开发各个阶段的一系列文档。

对于某些组织来说,RUP 之类的产品可能太过庞大了。如果开发环境多样,或者工程预算很紧张,这样的开发小组可能不想或者无法对基于软件的开发系统实现标准化。另外,学习曲线也可能是一个影响因素,新的工程人员如果对过程软件不熟悉,就必须学习如何使用它,同时还要尽快地了解产品和现有的代码基(这样势必使新加入的工程人员手忙脚乱)。

## 6.3 软件工程方法论

软件生命期模型提供了一种形式化方法来回答这样一个问题“下一步要做什么”,紧接着我们可能会问“下一步该怎么做呢?”,但是软件生命期模型很少能够回答这个问题,但诸如 RUP 的形式化系统除外。要对“怎么做”之类的问题做出回答,为此已经开发了许多方法论,这些方法论可以为专业的软件开发提供许多实用的经验。有关软件方法论的书和文章数不胜数,不过最近有两个新方法特别值得关注,即极限编程(Extreme Programming)和软件 Triage (Software Triage)。

### 6.3.1 极限编程 (XP)

几年前,本书作者之一下班回家时,告诉妻子他的公司采用了极限编程的某些原则。她开玩笑说“希望你能系上安全带”。尽管极限编程这个名字有些夸张,实际上它只是将既有的软件开发原则和新的原则捆绑在一起,而形成的一种越来越普及的全新方法论。

XP 因 Kent Beck 所著的《eXtreme Programming eXplained》(Addison-Wesley, 1999)一书推广开来,极限编程主张采用优秀的软件开发的最佳实践,并将其上升到一个新的高度。例如,测试是很有意义的,这一点大多数程序员都同意。在极限编程中,测试更被看作是一个好东西,以至于甚至会在编写代码之前先编写测试。

#### XP 理论

极限编程方法论由 12 条主要的指导原则组成。这些原则在软件开发过程的所有阶段都有体现,而且对工程人员每天完成的任务有直接的影响。

#### 根据要求做规划

在瀑布模型中,只会做一次规划,即在过程开始时规划。在螺旋方法中,规划则是每次迭代中的第一个阶段。在 RUP 中,规划是大多数工作流中少不了的一步。在极限编程中,规划不仅仅是一个步骤,

还是一个永不结束的任务。极限编程小组先从一个大致计划开始，其中只抓住所开发产品的几个重点。在开发过程中，这个计划会根据需要得到改进和修改。这样做的道理是，条件会不断地改变，而且会一直得到新的信息。

在极限编程中，由谁实现特性，就应由他对这个特性做出估计。这有助于避免一些不切实际的情况，否则，实现人员可能被迫遵守一个不可行的安排。最初，估计可能相当粗略，也许只是以周为单位来估计实现一个特性（功能）需要多长时间。随着时间越来越短，估计的粒度也越来越细。特性会分解为小任务，完成这些小任务应不超过 5 天。

### 建立小的发布版本

极限编程的一个理论是，如果软件项目一次想要做太多工作，那么风险就会更大，也更为笨重。极限编程不鼓励过大的软件发布，这会涉及核心修改，而且发布说明可能长达数页，XP 建议建立较小的发布版本，而且发布周期要接近两个月而不是长达 18 个月。由于发布周期这么短，只有最重要的特性才会放到产品中。这就要求工程部门和市场部门对于哪些特性确实重要达成一致。

### 采用共同的“隐喻”

XP 使用隐喻（metaphor）一词，而其他方法论可能使用体系架构（architecture）来表示同样的概念。其思想是，小组的所有成员都应当有一个共同的系统高层视图。这不必是对象如何交互或者 API 如何编写之类的具体问题。相反，隐喻是系统组件的高层模型。小组成员应当使用隐喻，这样在讨论项目时可以采用相同的术语。

### 简化设计

热衷 XP 的工程人员常念叨的口头禅是“要避免过分的一般性”。这与许多程序员的一般想法有所背离。如果要完成一个任务，如设计一个基于文件的对象库，可能会从这样一条路着手：即为所有基于文件的存储问题提供一个终极解决方案。设计可能很快会演变为涵盖多种语言和各类对象。XP 指出，应当向一般性的反方向倾斜。不要力图建立一个要夺大奖、受称赞的完美对象库，而应当设计最简单的对象库，只要能完成任务就行。你应当理解当前的需求，并编写代码来满足这些要求，而避免过于复杂的代码。

你可能不太习惯做简单的设计。取决于你做的是哪一类的工作，代码可能需要存在数年之久，而且可能会由代码的其他部分使用，对此你也许从未想到过。如第 5 章所述，如果要加入一些将来可能有用的功能，这就存在一个问题，你并不知道这些假想的用例是什么，而且很难得到一个好的设计。这就是一种过虑的情况。与此不同，XP 指出，应当建立对当前有用的东西，但要留有余地以便以后再做修改。

### 不断地测试

《eXtreme Programming eXplained》中指出，“如果没有一个相应的自动化测试，实际上任何程序特性（功能）都不能认为真的存在”。根据这种说法，极限编程把测试摆到了一个很高的位置。作为一个 XP 工程人员，职责之一就是编写代码的相应单元测试。单元测试往往是一小段代码，用以确定某一段功能能够正常工作。例如，对于一个基于文件的对象库，单元测试可能包括 testSaveObject、testLoadObject 和 testDeleteObject。

XP 则将单元测试更向前推进一步，建议应当在编写具体代码之前编写单元测试。当然，由于代码尚未编写，因此不可能通过测试。从理论上讲，如果测试是全面的，就应该能知道什么时候代码才算完成，因为此时所有测试都会顺利通过。可以指出，这就是“极限”。

### 必要时重构

大多数程序员会不时地重构（refactor）他们的代码。重构是一个重新设计既有工作代码的过程，从而考虑到新得到的知识，或者编写代码后才发现的其他方法。在传统的软件工程进度安排中很难加入重构，因为重构的结果不像实现一个新特性那样具体。不过，好的管理人员都会认识到，重构对于代码长

期的可维护性是相当重要的。

极限重构是指，在开发过程中识别出哪些情况下重构是有用的，并相应地完成重构。并不是在发布之初就确定产品现有的哪些部分需要设计工作，而是由 XP 程序员去学习如何在开发过程中找出可以重构的代码。尽管这种做法几乎总是会带来未预计和未安排的任务，但是在适当的时候重构代码会使特性的开发更为容易。

### 配对编写代码

配对编程 (pair programming) 概念作为一种人际互动 (touchy-feely) 软件过程可以算是极限编程的一个特色。实际上，配对编程往往比想像中还要实用。XP 建议所有成品代码都应当同时由两个人协作编写。显然，只有一个人能操控键盘 (即具体编写代码)，另一个人则站在更高层上，考虑诸如测试、必要的重构以及项目整体模型等问题。

举例来说，如果你负责为应用中一个特定功能编写用户界面，可能希望请该功能的原作者坐在一旁帮助你。他能建议你如何正确地使用这个功能，警告你要提防哪些“陷阱”，还可以在一个高层次上对你的工作进行监督。即使你得不到原作者的帮助，小组中的另一个成员也能提供帮助。道理很简单，通过配对工作可以共享双方的知识，保证正确的设计，并引入一个非正式的检查 and 平衡体系。

### 共享代码

在许多传统的开发环境中，代码所有权往往相当明确，而且会带来一定的限制。本书作者之一就曾经在这样一个环境中工作过，其中管理人员明确地禁止大家修改小组中其他成员编写的代码。XP 则走了另一个极端，声明代码为所有人所共有。程序员之所以会携起手来，向“固步自封”说再见，XP 的这个方面可谓功不可没。实际上，这不太算是“人际互动”。

出于多种原因，共同所有权很实用。从管理的角度看，一个工程人员突然离开的话，危害不会太大，因为还有其他人理解他的那部分代码。从工程人员的角度看，基于共同所有权，可以对系统如何工作建立一个共同观点。这有助于设计任务，而免除了单个程序员的负担，因为不会要求他做出对整个项目都有意义的修改。

对于共同所有权，需要说明很重要的一点，无需每一位程序员都熟悉每一行代码。更正确的理解是，项目是整个团队的共同努力结果，没有必要让某一个人了解全部内容。

### 不断地集成

所有程序员都不会对“可怕的”代码集成工作感到陌生。当你发现所看到的对象库并没有按原来编写的那样工作时，就要完成这个集成任务了。子系统聚在一起时，问题就会暴露出来。XP 认识到这种现象的存在，并鼓励在开发代码的过程中经常性地代码集成到项目中。

XP 推荐了一种用于集成的特定方法。两个程序员 (开发代码的配对) 在指定的“集成点”完成代码的合并。在代码 100% 地成功通过测试之前，此代码不能提交。由于有一个集成点，这可以避免冲突，而且集成可以清楚地定义为一个必须在提交之前出现的步骤。

我们还发现了一个类似的方法，采用这种方法，单个程序员也能完成集成。工程人员在把代码提交到代码库之前，要单独或配对地运行测试。由一台指定机器持续地运行自动化测试。当自动化测试失败时，开发小组会收到一个电子邮件，指出存在的问题，并列出现最近提交的代码。

### 合理的工作时间

XP 对投入的时间也有涉及。据称，如果程序员休息得好，不仅心情愉快，而且效率更高。XP 建议一周工作大约 40 小时，并警告不要过度工作连续两周以上。

当然，不同的人需要多少休息存在差异。不过，重点是，如果你坐下来编写代码，但头脑不清醒，写出来的代码肯定不好，而且也会违反许多 XP 原则。



### 有一个顾客在场

由于崇尚 XP 的工程组会不断地改进其产品计划，而且只在产品中加入对当前而言必要的东西，因此如果有一个顾客能够参与会很有意义。尽管往往很难说服一个顾客真正在开发过程中一直在场，但是工程小组与最终用户之间应当保持交流，这一点无疑很重要。除了能够帮助设计各个特性之外，顾客还能根据各自的需要确定哪些任务更加优先。

### 共享共同的编码标准

由于存在共同所有权原则，而且实践中会采用配对编程方法，如果每个工程人员都有自己的一套命名和标识约定，那么在极限环境中编写代码就会很困难。XP 并不特别推崇某一种风格，不过指出了——一个指导原则：如果看一段代码的时候能够很容易地发现其作者是谁，开发小组就很有可能需要定义一种更好的编码标准。

### XP 实践

XP 倡导者称，极限编程的 12 个原则彼此关联，如果采用其中的某些原则而不采用另外的一些原则，这会大大影响这个方法论的实施。例如，配对编程对测试就至关重要，因为如果你不能确定如何测试一段特定代码，你的同伴可以助一臂之力。另外，如果你哪一天太累，想要直接跳过测试，你的同伴则可以在一旁唤醒你的理智。不过，有些 XP 指导原则可能很难实现。对于某些工程人员来说，在编写代码之前编写测试，这种想法就很抽象。对他们而言，在有代码可供测试之前，只要设计出测试就足够了，此时不用具体编写测试。许多 XP 原则定义很严格，不过如果你了解基本理论，就能发现可以对这些原则稍做调整，来满足项目的需要。

XP 的协作方面也很有难度。配对编程的好处可能很多，但是管理人员也许很难合理地安排一半人具体编写代码。小组中的有些成员可能不喜欢这种密切协作，在自己写代码时，有别人在旁边看着，可能会让他觉得不舒服。如果开发小组中的成员分布得很开，或者需要定期地远程通信，显然很难做到配对编程。

对于有些组织来说，极限编程可能太极端了。一些规模很大的公司对于工程开发往往有正式的条文，这些公司在采纳像 XP 之类的新方法时可能就很慢。不过，即使你的公司拒绝采用 XP，如果能理解其基本原理，也能提高自己的工作效率。

## 6.3.2 软件 triage

在一本名字很恐怖的书《Death March》(Prentice Hall, 2003) 中，Edward Yourdon 介绍了导致软件进度滞后、人员紧缺、预算超支或设计糟糕的一些常见的危险条件。Yourdon 的理论称，软件项目如果处在这种状态，即便是当前最好的软件开发方法论也无济于事。在本章中你已经了解到，许多软件开发方法都是建立在形式化文档基础上的，或者采用一种以用户为中心的方法来设计。如果项目已经处在“死亡之旅”中，就没有时间、没有机会来实施这些方法了。

软件 triage 的基本思想是，如果一个项目已经处在很糟糕的状态下，资源就会紧缩。不仅时间不够，工程人员不够，资金也很短缺。当项目进度滞后时，管理人员和开发人员需要克服的一个最大障碍是：可能无法在预定的时间内满足最初提出的需求。原本要完成的任务只好简单地列在“必须有的功能”、“应当有的功能”和“最好有的功能”列表中，等待以后实现。

软件 triage 是一个麻烦而且细致的过程。通常要求经历过“死亡之旅”项目的有经验的老手指导，并做出艰难的抉择。对工程人员来说，最重要的一点是，某些情况下可能需要将我们熟悉的一些过程抛在一边（遗憾的是，这样可能也会丢掉一些原有的代码），以便项目如期完工。

## 6.4 建立自己的过程和方法论

肯定有一种软件开发方法论是我们全心拥护的，而这不一定非得是上述的某一种方法论。不管是哪

一本书或哪一种工程理论，一般不太可能刚好满足你的项目或组织的需要。建议你尽可能多地了解多种方法，并设计自己的过程。结合不同方法中的概念要比完全让自己去想容易得多。例如，RUP 可以支持一种类 XP 的方法。以下是建立自己的软件工程过程的一些提示。

#### 6.4.1 以开放的心态接纳新思想

有些工程技术乍看上去可能很不可思议，或者好像不可行。要把软件工程方法论中的创新看作是改进现有过程的一个途径。在可能的情况下要尽量尝试。如果极限编程听起来很有趣，但不能保证它确实能在你所在组织中奏效，就可以慢慢地看它是否可行，一次采纳几个原则，或者在一个较小的预研项目中先做尝试。

#### 6.4.2 汇总新思想

更常见的情况下，工程小组由来自各个不同背景的人所组成。这些人可能包括资深的老手，经验丰富的顾问，刚刚毕业的研究生以及一些博士。每个人都有不同的经验，而且对于软件项目应当如何运行也都有自己的一套想法。有时，通过将适用于各个不同环境的工作方式加以结合，往往能得到最佳的过程。

#### 6.4.3 明确哪些可行，哪些不可行

在项目最后（或者更好情况下，可以在项目进行过程中），要把小组召集在一起对过程做出评估。有时只有当整个小组停下当前的工作，共同思考一个问题，才会发现重要的问题。也许每个人都知道一个问题的存在，但是从未有人提起过！要考虑哪些是不可行的，并明确这些部分能否修正。有些组织在提交任何源代码之前需要正式的代码审查。如果代码审查需要太长时间，太过枯燥，那么谁的工作也做不好，大家应当在一起讨论代码审查技术。另外，还要考虑哪些方面工作得很好，并了解这些部分如何扩展。例如，可以建立一个网站来维护各种任务（小组能够编辑这些任务），这种想法如果可行，那么可以多花一些时间把这个网站建得更好。

#### 6.4.4 不要做叛逃者

不论过程是由管理人员控制，还是由小组自行建立，总有它存在的原因。如果你的过程涉及编写正式的设计文档，就要确保文档由你编写。如果你认为这个过程很糟糕，或者太过复杂，可以看看是不是能与管理人员做些讨论。不要一味地躲避这个过程，它还是会“找上门来”的。

### 6.5 小结

本章介绍了软件过程的一些模型和方法论。建立软件还有许多其他的方法，这包括一些正式和非正式的方法。开发软件时，可能没有一种完全正确的方法，而只有最适合你的方法。发现这种方法的最佳途径是自己做研究，尽可能地从多种不同方法中学习，向别人讨教他们在使用各种方法时的经验，并迭代地完成过程。要记住，分析一个过程方法论时，有一个真正有意义的标准，这就是看这种方法论对你的小组编写代码有多大的帮助。

到本章为止，本书第一部分就结束了，这一部分所讨论的都是有关软件设计方面的内容。你了解了如何设计一个程序，如何组织对象关系，如何利用既有的模式和库，如何与别人一同有效地编写代码，如何管理软件开发过程等等。在本书余下的部分中，你学到的这些设计原则将与 C++ 紧密结合。下一部分将深入到细节当中，介绍如何用 C++ 编写专业质量的代码。在深入到书中有关编写代码的部分中时，不要忘记了前面几章学到的设计原则。我们之所以把有关设计的章节放在最前面，就是想要突出强调它们的重要性。



# 第二部分

## 编写 C++ 代码方式

### 第 7 章 好的编码风格

如果你打算每天花几小时坐在键盘前面编写代码，那你应该写点像样的东西出来，让自己能够引以为豪。编写代码完成某个任务，这只是程序员的一部分工作而已。毕竟，除了程序员，任何人都能掌握编写代码的基本要领。只有真正参透了编程才会有自己的编码风格。

本章要讨论从哪些方面来看代码的好坏。在介绍的过程中，还将了解到建立 C++ 风格的几种方法。你会发现，只是对代码风格有所调整，就能使代码大不相同。例如，Windows 程序员编写的 C++ 代码通常都有自己的风格，他们会采用 Windows 的惯例。如果是 Mac OS 程序员编写 C++ 代码，情况就全然两样，就好像使用的是另外一种完全不同的语言一样。适当地接触多种不同的风格，有助于增长见识，这样当打开一个 C++ 源文件时，即使它与你所了解的 C++ 代码不太像，也不至于太灰心、太茫然。

#### 7.1 为什么代码看上去要好

要想编写出风格“好”的代码，是需要花时间的。你可能只需几个小时就能把一个 XML 文件的程序解析成纯文本文件。但是如果考虑到功能分解，加上适当的注释，而且要求有清晰的结构，那么编写同样的一个程序可能需要花费数天之久。这到底值不值得呢？

##### 7.1.1 提前考虑

如果一年之后，有一个新的程序员要来使用或者维护你的代码，你对自己的代码有多少信心呢？本书作者之一就有这样一次经历，在开发一个 Web 应用时，Web 应用的代码越来越混乱，这就充分表明他的开发小组应当提前考虑，能够假想一年之后会有新人介入。如果没有任何文档，而且函数也很吓人，动不动就长达好几页，那么这个可怜的人怎么能迅速赶上，了解原先的代码基呢？在编写代码的时候，应该想像将来可能会有其他的某个人要维护这些代码。你自己能记住这些代码是如何工作的吗？如果你无法亲自提供帮助该怎么办？如果代码写得好，这些问题就可以避免，因为好代码不仅容易阅读，而且还很好理解。

##### 7.1.2 保持清晰

之所以要编写好代码，还有一个原因，这就是这种代码本身就可以提供问题的解答。除非你独自开

发一个项目，而且以后也是如此，否则肯定有其他程序员要查看你的代码，甚至还有可能修改你的代码。如果你的小组确实能读懂你编写的代码，他们就不会老问你问题，也不会对你抱怨多多。

### 7.1.3 好的代码风格包括哪些因素

要逐一列出哪些特性可以确定代码有“好”的风格，这往往很困难。经过一段时间的历练，会找到自己喜欢的风格，而且会在其他人写的代码中发现一些有用的技术。也许更重要的是，你会遇到一些糟糕的代码，这些代码会教会你哪些是应该避免的。不过，好代码都有一些普遍原则，本章将分别讨论这些原则。

- 文档
- 分解
- 命名
- 语言使用
- 格式化

## 7.2 为代码加注释

在编程领域中，文档通常是指源文件中包含的注释。注释能告诉别人你在写相应的代码时在想些什么。如果有些信息从代码本身不能一眼看出，就可以放在注释里。

### 7.2.1 写注释的原因

写注释是一个好的想法，这一点可能显而易见，不过为什么需要对代码加注释，你是不是一直在考虑这个问题？有时程序员能够认识到加注释的重要性，但是并没有充分理解为什么注释很重要。对此有很多原因，以下将分别说明。

#### 加注释来解释用法

使用注释的一个原因是，它可以解释客户应当如何与这段代码交互。在第 5 章已经提到过，头文件中每一个可以公共访问的函数或方法都应当有一个注释，用以解释这个方法或函数要做些什么。有些组织倾向于建立一种形式化的注释“模板”，即注释都采用相同的格式，分别列出各个方法的用途，有哪些参数，返回值是什么，可能抛出哪些异常等等。

为公共方法提供注释，这样做有两个作用。首先，对于在代码中无法表述的东西，你有机会用自然语言（如英语）来表述。例如，C++ 代码中确实没有办法指出一个媒体播放器对象的 `adjustVolume()` 方法只能在调用了该对象的 `initialize()` 方法之后才能调用。不过，完全可以在注释中指出这种限制，如下所示。

```
/*  
 * adjustVolume()  
 *  
 * Sets the player volume based on the user's  
 * preferences  
 *  
 * This method will throw an "UninitializedPlayerException"  
 * if the initialize() method has not yet been called.  
 */
```

公共方法的注释还有一个作用，它可以表述用法信息。C++ 语言要求必须指定方法的返回类型，不过它并没有提供一种方法来说明返回值的实际含义。例如，`adjustVolume()` 方法的声明可能指出它要

返回 int，但是读到这个声明的客户并不知道 int 表示的是什么。在注释中还可以包括其他一些辅助数据，如下所示。

```
/*
 * adjustVolume()
 *
 * Sets the player volume based on the user's
 * preferences
 *
 * Parameters:
 *   none
 * Returns:
 *   an int, which represents the new volume setting.
 *
 * Throws:
 *   UninitializedPlayerException if the initialize() method has not
 *   yet been called.
 */
```

### 加注释来解释复杂的代码

在具体的源代码中，好的注释也很重要。如果有一个程序要处理用户输入，并向控制台写出一个结果，在这个简单的程序中，所有代码可能都很容易阅读和理解。不过，在专业领域，往往需要编写算法很复杂的代码，或者这些代码过于深奥，仅凭查看可能根本无法理解。

请考虑以下的代码。这段代码写得并不坏，不过可能无法马上了解它要做什么。如果你以前见过这种算法，可能很快就能理解，但是如果一个新来的人看到这段代码，也许不能理解代码是如何工作的。

```
void sort(int inArray[], int inSize)
{
    for (int i = 1; i < inSize; i++) {
        int element = inArray[i];

        int j = i - 1;
        while (j >= 0 && inArray[j] > element) {
            inArray[j+1] = inArray[j];
            j--;
        }
        inArray[j+1] = element;
    }
}
```

更好的做法是通过增加注释来介绍所用的算法。对上述函数修改后，最上面就有一个很全面的注释，它在一个高层次上解释了算法，对于可能搞糊涂的具体代码行，则通过内联（内部）注释来解释，如下所示。

```
/*
 * Implements the "insertion sort" algorithm. The algorithm separates the array
 * into two parts--the sorted part and the unsorted part. Each element, starting
 * at position 1, is examined. Everything earlier in the array is in the sorted
 * part, so the algorithm shifts each element over until the correct position is
 * found for the current element. When the algorithm finishes with the last
 * element, the entire array is sorted.
 */
void sort(int inArray[], int inSize)
{
    // Start at position 1 and examine each element.
    for (int i = 1; i < inSize; i++) {
```

```

int element = inArray[i];

// j marks the position in the sorted part of the array.
int j = i - 1;
// As long as the current slot in the sorted array is higher than
// the element, shift the slot over and move backwards.
while (j >= 0 && inArray[j] > element) {
    inArray[j+1] = inArray[j];
    j--;
}
// At this point the current position in the sorted array
// is *not* greater than the element, so this is its new position.
inArray[j+1] = element;
}

```

这段新代码确实比原来的长多了，但是，如果读代码的人原先不了解排序算法，通过增加注释，他就更有可能理解这个函数。有些组织不主张采用内联注释。在这种情况下，至关重要是既要编写清晰的代码，又要在函数的上面提供好的注释。

#### 加注释来表达元信息

使用注释的另一个原因是可以提供一个更高层次上的信息，而不是代码本身的信息。这种元信息 (metainformation) 提供了有关创建代码的详细信息，但不涉及其行为的具体细节。例如，你的组织可能想记录各个方法的原作者。另外还可以使用元信息来引用外部文档，或参考其他代码。

下面的例子显示了几种元信息，包括文件的作者、文件创建的日期、所实现的具体特性等。其中还包括一些表述元数据 (元信息) 的内联注释，如某行代码解决了哪一个 bug (bug 编号)，还提示出以后在代码中可能会遇到的一个问题。

```

/*
 * Author: klep
 * Date: 040324
 * Feature: PRD version 3, Feature 5.10
 */
int adjustVolume()
{
    if (fUninitialized) {
        throw UninitializedPlayerException();
    }

    int newVol = getPlayer()->getOwner()->getPreferredVolume();

    if (newVol == -1) return -1; // Added to address bug #142 - jsmith 040330

    setVolume(newVol);

    // TODO: What if setVolume() throws an exception? - akshayr 040401

    return newVol;
}

```

很容易对注释过于热衷。对此有一个好的做法，应讨论哪几类注释对你的小组最有用，并建立相关的策略。例如，小组中的一个成员使用了一个“TODO”注释来指示还需要调整的代码，但是别人都不了解这种约定，他们不知道“TODO”注释指出了下一步要做的工作，就往往会忽视本来需要增加的代码。

如果你的小组决定使用元信息注释，要确保所有人都包括了相同的信息，否则文件就会不一致！



### 7.2.2 注释风格

在为代码加注释时，每个组织都有自己不同的方法。有些环境中，会强制采用一种特定的风格，提供建立代码文档的一种公共标准。在另外一些情况下，注释的质量和风格全由程序员决定。以下例子展示了几种加代码注释的方法。

#### 对每一行都加注释

为了避免文档不足的情况，一种做法是为每一行都加注释，这就要求建立充分的文档。对每一行代码都加注释，这可以确保所写的每行代码都有其特定的原因。在实际中，如果有如此之多的注释，代码往往不能很好地扩展，不仅很乱，而且很乏味。例如，请考虑以下注释，这些注释就很没用。

```
int result;           // Declare an integer to hold the result.

result = doodad.getResult(); // Get the doodad's result.

if (result % 2 == 0) { // If the result mod 2 is 0 . . .
    logError();        // then log an error.
} else {              // otherwise . . .
    logSuccess();      // log success.
}                    // End if/else

return (result);      // Return the result
```

这段代码中的注释只是把每一行代码换种方式复述一遍，就好像要用自然语言讲一个很容易读懂的故事一样。如果读代码的人至少有基本的 C++ 技能，这就根本没有意义。这些注释没有为代码增加任何额外的信息。具体地，请看以下这行代码：

```
if (result % 2 == 0) { // If the result mod 2 is 0 . . .
```

这个注释只是用英语对代码做了一个翻译。它并没有指出程序员为什么要对结果 (result) 和值 2 使用取模 (mod) 操作符。更好的注释应当是：

```
if (result % 2 == 0) { // If the result is even . . .
```

尽管修改后的注释在大多数程序员看来也是很显然的，但它确实提供了有关代码的一些额外信息。之所以结果要模 2，这是因为代码需要查看结果是否为偶数。

尽管这种做法可能太过周密，有些多余，但是在有些情况下充分注释非常有用，否则代码将很难理解。以下代码也对每一行都加了注释，但是这些注释确实很有帮助。

```
// Call the calculate method with the default values.
result = doodad.calculate(getDefaultStart(), getDefaultEnd(), getDefaultOffset());

// To determine success or failure, we need to bitwise AND the result with the
// processor-specific mask (see docs, page 201).
result = result & getProcessorMask();

// Set the user field value based on the "Marigold Formula."
setUserField( (result + kMarigoldOffset) / MarigoldConstant + MarigoldConstant );
```

尽管这个代码是从一个特定上下文中节取的，但是通过注释，能很好地了解每一行要做什么。如果没有这些注释，将很难理解涉及 & 的计算和神秘的“Marigold Formula”。

对代码中的每一行都加注释，往往是做不到的，不过，如果代码太复杂，确实需要有如此充分的注释，你也要注意，不要只是把代码用英语翻译一遍而已，也就是说，不要只是解释发生了什么，而应当指出为什么要这样做。

### 前缀注释

你的小组可能决定在所有源文件前面都加上一个标准注释。这是一个绝好的机会，可以在这里描述有关程序和特定文件的重要信息。在每个文件最前面的“文档”中，你可能希望加入以下信息：

- 文件/类名
- 最后一次修改时间
- 原作者
- 文件所实现特性的编号（特性 ID）
- 版权信息
- 文件/类的简要描述
- 未完成的特性
- 已知的 bug

你所用的开发环境可能允许创建一个模板，可以自动地在新文件前面加上前缀注释（prefix comment）。有些源代码控制系统甚至还能帮助填入元数据，如并发版本系统（Concurrent Versions System, CVS，也称协同版本系统）。例如，如果注释中包含串 \$Id\$, CVS 会自动地扩展这个注释，其中包括作者、文件名、修改版本和日期。

以下显示了一个前缀注释的例子。

```
/*
 * Watermelon.cpp
 *
 * $Id: Watermelon.cpp,v 1.6 2004/03/10 12:52:33 klep Exp $
 *
 * Implements the basic functionality of a watermelon. All units are expressed
 * in terms of seeds per cubic centimeter. Watermelon theory is based on the
 * white paper "Algorithms for Watermelon Processing."
 *
 * The following code is (c)copyright 2004, FruitSoft, Inc. ALL RIGHTS RESERVED
 */
```

### 固定格式的注释

编写采用标准格式的注释，以供外部文档生成器解析，这是一种越来越流行的编程实践。在 Java 语言中，程序员可以采用一种标准格式编写注释，以便一种称为 JavaDoc 的工具为项目自动地创建超链接文档。对于 C++ 来说，也有这样一个免费工具，名为 Doxygen（可以从 [www.doxygen.org](http://www.doxygen.org) 得到），这个工具可以解析注释，并自动生成 HTML 文档、类图、UNIX 手册主页和其他有用的文档。Doxygen 甚至还能识别并解析 C++ 程序中 JavaDoc 风格的注释。

```
/**
 * Implements the basic functionality of a watermelon
 *
 * TODO: Implement updated algorithms!
 */
class Watermelon
{
public:
    /**
     * @param initialSeeds The starting number of seeds
     */
}
```

```
*/  
Watermelon(int initialSeeds);  
  
/**  
 * Computes the seed ratio, using the Marigold  
 * algorithm.  
 *  
 * @param slowCalc Whether or not to use long (slow) calculations  
 * @return The marigold ratio  
 */  
double calcSeedRatio(bool slowCalc);  
};
```

Doxygen 能够识别 C++ 语法和特殊的注释指令（如 @param 和 @return），从而生成可定制的输出。图 7-1 显示了一个用 Doxygen 生成的 HTML 类参考文档的例子。

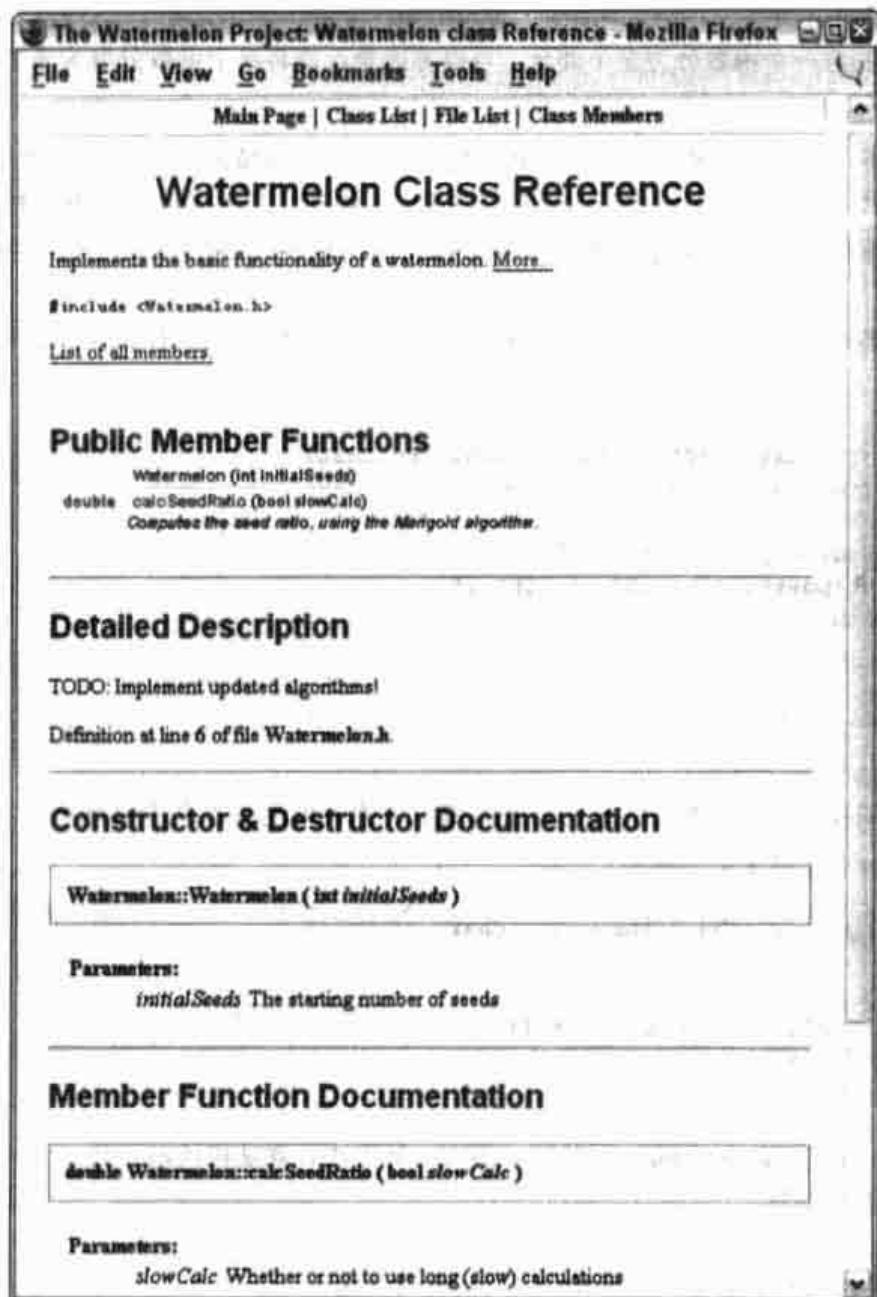


图 7-1

自动生成的文档（如图 7-1 所示的文件）在开发中可能很有帮助，因为开发人员可以由此获得对类及其关系的一个高层描述。你的小组可以很容易地定制像 Doxygen 这样的工具，使之适应于你采用的注释风格。理想情况下，你的小组可以专门用一台机器每天只是生成文档。

### 特殊 (Ad hoc) 注释

大多数情况下，都是根据需要来使用注释。以下是对在代码体中出现的注释的一些指导原则。

- 尽可能地避免有攻击性或贬低性的语言。你无法知道将来谁会看你的代码。
- 开一些内部玩笑通常是允许的。但要经过管理人员的批准。
- 尽可能地引用 bug 编号或特性 ID。
- 如果觉得将来可能有人希望根据注释与你取得联系，请加上你的基本信息和日期。
- 不要冒然加上别人的基本信息和日期，以免为此承担责任。
- 要记住更新代码时一定要更新注释。如果代码文档详尽，但是提供的信息有误，再没有比这更让人糊涂的了！
- 如果使用注释将一个函数分为多个部分，可以考虑是否能将这个函数分解为多个更小的函数。

### 自描述文档

写得好的代码不一定需要大量的注释。最好的代码要写得很容易阅读。如果发现要为每一行都增加一个注释，就要考虑一下能不能重写代码，让它更好地实现你在注释里所说的工作。要记住 C++ 是一种语言。它的主要用途是告诉计算机怎么做，不过还可以利用这种语言的语义向读代码的人解释代码的含义。

复制 C 风格字符串的函数实现就是一个很典型的例子。以下代码没有任何注释，不过也不需要任何注释。

```
void copyString(const char* inSource, char *outDest)
{
    int position = 0;

    while (inSource[position] != '\0') {
        outDest[position] = inSource[position];
        position++;
    }

    outDest[position] = '\0';
}
```

以下代码也做同样的工作，不过这段代码太简练了，读者可能很难马上就理解。这个实现本身并没有错，不过它需要一些注释来做出解释。

```
void copyString(const char* inSource, char* outDest)
{
    int i = 0;
    while (outDest[i] = inSource[i++]);
    outDest[i] = '\0';
}
```

要编写自描述 (self-documenting) 的代码，还有一种方法，就是将代码分解 (decompose) 为更小的部分。下面将详细介绍分解。

好代码自然是可读的，只需注释来提供额外的有用信息。

### 7.2.3 本书中的注释

本书中，你将看到的代码示例通常会利用注释来解释一些复杂的代码，或者指出一些不那么明显的地方。为节省篇幅，我们一般没有加任何前缀注释和固定格式注释，不过强烈建议你在专业的 C++ 项目中包含前缀注释和固定格式注释。

## 7.3 分解

分解就是将代码分解为较小部分的实践做法。在代码编写领域中，如果打开一个文件，发现函数多达 300 行代码，而且存在大量嵌套的代码块，再没有比这更恐怖的了。理想地，每个函数或方法都应当只完成一个任务。如果子任务过于复杂，就应当分解为单独的函数或方法。例如，如果有人问你某个方法要做什么，而你这样回答“首先，它做 A，然后做 B，接下来，如果 C 则做 D，否则它要完成 E。”如果是这样，可能应该分别为 A、B、C、D 和 E 建立单独的辅助方法。

分解存在着模糊性。有些程序员可能认为所有函数都不应超过一页。这可能是一个不错的经验，不过你很可能发现一段只有 1/4 页的代码也确实需要分解。另外还有一条经验，不论代码是长还是短，如果粗略地看代码，只看代码格式，不看代码内容，其中不应该存在过于稠密的区域。例如，我们故意对图 7-2 和图 7-3 做了模糊处理，使你看不清它们的内容。显然，图 7-3 中的代码比图 7-2 所示的代码得到了更好的分解。



图 7-2

### 7.3.1 通过重构来分解

你可能精力充沛，而且确实乐于编程，你会很快地写出代码，尽管最终代码确实能达到目的，但是由于写得太快，往往称不上是一个好代码。所有程序员都时不时会做这种事情。有时这种短期的全力编写代码是项目中生产力最高的阶段。

修改代码时也可能产生“稠密”代码。随着新需求的出现，以及对 bug 做出修正，现有代码中会充斥着一些小的修改。计算领域中的 *cruft* 一词就是指，尽管每次只调整少量代码，但是日积月累，最终会使一段原本不错的代码变成一堆补丁和特例。

不论代码从一开始就是一段不可读的稠密 *cruft*，或者只是经过一系列修改演变成这样的，都很有



必要重构 (refactoring)，从而周期性地肃清代码中堆积的补丁。通过重构，会重新考虑现有代码，把它重写为更可读、更可维护的代码。重构也是一个重新考虑代码分解的机会。如果代码的用途有变，或者刚开始根本未做分解，就可以在重构代码的时候泛泛地看看代码，确定是否需要把它分解为更小的部分。

```

// 函数 getInputs() 从用户处获取输入，并将输入存储在变量中
// 函数 performCalculations() 根据输入计算风速和气压
// 函数 outputResults() 输出计算结果

// 函数 getInputs()
void getInputs() {
    // 获取风速
    cout << "Enter wind speed: ";
    double windSpeed;
    while (cin.get() != '\n')
        continue;
    cin >> windSpeed;

    // 获取气压
    cout << "Enter barometric pressure: ";
    double barometricPressure;
    while (cin.get() != '\n')
        continue;
    cin >> barometricPressure;

    // 获取海拔
    cout << "Enter altitude: ";
    double altitude;
    while (cin.get() != '\n')
        continue;
    cin >> altitude;

    // 计算风速和气压
    cout << "Wind speed: " << windSpeed << endl;
    cout << "Barometric pressure: " << barometricPressure << endl;
    cout << "Altitude: " << altitude << endl;
}

// 函数 performCalculations()
void performCalculations() {
    // 计算风速和气压
    double windSpeed = 0;
    double barometricPressure = 0;
    double altitude = 0;

    // 计算风速和气压
    cout << "Wind speed: " << windSpeed << endl;
    cout << "Barometric pressure: " << barometricPressure << endl;
    cout << "Altitude: " << altitude << endl;
}

```

图 7-3

### 7.3.2 根据设计来分解

对于喜欢拖延的人来说，分解还有一个好处。如果从一开始就采用分解的思路编写代码，就会把难的部分向后拖延。这种编码风格通常称为自顶向下设计 (top-down design)，即首先对程序有一个高层视图，然后逐步地移向更为特定的部分。

例如，使用自顶向下设计，可以马上为一个模拟龙卷风的程序确定代码主体。以下代码显示了这个程序的一个可能的 main() 实现。

```

int main(int argc, char** argv)
{
    cout << "Welcome to the Hurricane Simulator" << endl;

    getUserInputs();
    performCalculations();
    outputResults();
}

```

采用自顶向下方法，可以做到两点。首先，可以立即开始编写代码。尽管最后得到的程序与最初确定的高层视图可能有出入，但编写一些代码起码可以帮助你整理思路。其次，程序会很自然地沿着一条逐步分解的道路发展。在编写代码时可以将每一个特性都完整地实现，但也可以在实现每个方法或函数时，都考虑要把其中的哪些部分延后处理，这样一来，与前一种做法相比，程序通常不会太稠密，而且更易于组织。

当然，我们还是建议你在实际开始编写代码之前要对程序先做一些设计。不过，在确定程序中某一部分的具体实现时，或者在完成小项目时，自顶向下方法通常很有帮助。

### 7.3.3 本书中的分解

在本书许多例子中都将看到分解。在许多情况下，我们会提到一些方法，但是没有显示这些方法的实现。因为这些方法与相应例子关系不大，而且要占太多篇幅。

## 7.4 命名

计算机并不关心你对变量和函数如何命名，只要这些变量名和函数名不与另外的变量或函数发生冲突就行。之所以要命名，只是为了帮助你和你的同事处理程序中的单个元素。从这个目的来看，不免会对一种现象感到奇怪，即程序员们经常会在程序中使用不明确或不合适的名字。

### 7.4.1 选择一个好名字

变量、方法、函数或类的名字若能准确地描述其用途，这样的名字就是最好的。名字还可以提供额外的一些信息，如类型或具体用法。当然，要看名字好坏与否，真正的试金石是其他程序员是否能了解你通过这个名字想要表达些什么。

对于如何命名，并没有固定的规则，只是有一些对你的组织适用的规则。不过，有一些名字一般都不合适的。表 7-1 显示了有关命名的两个极端，可以看到一些很好的名字和一些很不好的名字。

表 7-1

很好的名字	不好的名字
srcName, dstName 很好地区别了两个对象	thing1, thing2 太过一般化
gSettings 表达出这是全局变量	globalUserSpecificSettingsAndPreferences 太长了
mNameCounter 表达出这是数据成员	mNC 太短，太含糊
performCalculations() 简单、准确	doAction() 太过一般化，不准确
mTypeString 很容易查看	_typeSTR256 这个名字除了计算机可能谁都不会喜欢
mWelshRarebit 很好地利用了内部玩笑	mIHateLarry 不合适的内部玩笑（有贬损意思）



### 7.4.2 命名约定

选择名字时一般不需要太多考虑，也不需要太多创新。在许多情况下，你可能想使用标准的命名技术。以下各类数据就可以采用标准命名。

#### 计数器

在你的编程生涯中，最早可能见过使用变量“i”作为计数器的代码。我们习惯于使用 i 和 j 分别作为（外循环）计数器和内循环计数器。不过，要当心嵌套循环。有一个常见的错误，有时实际上想表示第 j 个元素，却引用了第 i 个元素。有些程序员倾向于使用形如 `outerLoopIndex` 和 `innerLoopIndex` 之类的计数器。

#### 获取方法和设置方法

如果类包含一个数据成员，如 `mStatus`，习惯上会通过名为 `getStatus()` 的获取方法和名为 `setStatus()` 的设置方法来提供对这个成员的访问。C++ 语言没有为这些方法规定名字，不过，你的组织可能会采用诸如此类的一种命名机制。

#### 前缀

许多程序员在变量前面都加上一个字母，来提供一些有关变量类型或用法的信息。表 7-2 显示了一些常用的前缀。

表 7-2

前 缀	示 例 名	前缀字面含义	用 法
m _	mData _data	“成员 (member)”	类中的数据成员。有些程序员使用 _ 作为前缀来表示一个成员。另外一些程序员则认为 m 更可读
s	sLookupTable	“静态 (static)”	静态变量或数据成员。用于针对类的变量（即类变量，这些静态变量由所有类实例所共享）
k	kMaximumLength	“常量 (konstant)”（这是德语中的常量，还是拼错了？由你自己决定）	指示一个常量值。有些程序员还会用全大写的名字来指示常量
f	fCompleted	“标志 (flag)”	指示一个布尔值。特别适用于指定类的一个 yes/no 属性，可以基于这个属性值来改变对象的行为
n mNum	nLines mNumLines	“数字 (number)”	用作为一个计数器的数据成员。由于“n”看上去与“m”很接近，有些程序员会使用 mNum 作为前缀，mNumLines 就是如此
tmp	tmpName	“临时 (temporary)”	指示一个变量仅用于临时地保存一个值。由此反映出后面的代码不应依赖于这个变量的值

#### 大小写

代码中名字的大小写如何指定，对此有许多不同的方法。与编码风格中的大多数方面一样，最重要的是你的小组要采用一种标准的方法，而且所有成员都要遵循这种方法。如果一些程序员命名类时全用小写，并用下划线表示单词之间的空格（如 `priority_queue`），而另外一些程序员将类名中每个词的首字母大写（如 `PriorityQueue`），这无疑会导致代码的混乱。变量和数据成员一般都以小写字母开头，可以使用下划线来分隔不同单词（`my_queue`），也可以将单词的首字母大写来区分单词（`myQueue`）。

在 C++ 中，函数和方法通常都以大写字母开头，不过，可以看到，本书中我们采用了另一种风格，即使用小写的函数和方法名，以此来区别类名。不过与类名和数据成员名一样，在函数和方法名中，也

是通过单词首字母大写来分隔不同单词。

### 智能常量

假设你在编写一种带图形用户界面的程序。这个程序有许多菜单，包括文件（File）、编辑（Edit）和帮助（Help）。为了表示各个菜单的 ID，你可能决定使用一个常量。对于表示 Help 菜单 ID 的常量，kHelp 就是一个很合理的名字。

kHelp 这个名字一般情况下都算不错，不过如果有一天向主窗口中增加了一个 Help 按钮，就会有问题了。你需要一个常量表示这个按钮的 ID，但是 kHelp 已经被菜单用了。

要解决这个问题，有许多方法。一种方法是将两个常量放在不同的命名空间中，这在第 1 章曾经介绍过。不过，对于这么一个小问题，只不过是两个常量之间的命名冲突，采用命名空间好像有些大材小用了。可以将常量改名为 kHelpMenu 和 kHelpButton，这就能很容易地解决问题。不过，对常量命名还有一种更聪明的办法，就是把名字倒过来变成 kMenuHelp 和 kButtonHelp。

名字倒过来后，乍看上去好像有些不顺。不过，这样做有很多好处。首先，如果按字母表顺序列出所有常量，那么所有菜单常量会列在一起。如果开发环境提供有自动完成功能或弹出菜单（译者注：即键入属性或方法的部分名时，开发环境会自动选定完整的属性名或方法名，或者会弹出一个菜单，其中列出所有与之部分匹配的名字），在键入代码时，利用这种功能或者这样一个弹出菜单，就会方便许多。其次，这样可以提供一个较弱的命名体系，但是使用很容易。这里不是使用命名空间（因为命名空间可能会变得过于庞大），而实际上是把命名空间作为名字中的一部分。如果要表示帮助菜单中的单个菜单项，甚至还能进一步扩展这个层次体系，如可以使用常量名 kMenuFileSave。

### 匈牙利记法

匈牙利记法（Hungarian Notation）是一种变量和数据成员命名约定，Microsoft Windows 程序员对此都很熟悉。其基本思想是，不使用单字母前缀（如 m），而应当使用更详细的前缀来指示额外信息。以下代码行显示了匈牙利记法的使用。

```
char* pszName; // psz means "pointer to a null-terminated string"
```

匈牙利记法之所以得名，是因为它的创始人 Charles Simonyi 是一个匈牙利人。有人也指出，这也准确地反映出一个事实，使用匈牙利记法的程序最后看上去就像是用一种外国语言写的一样。出于后一种原因，有些程序员不喜欢采用匈牙利记法。在本书中，我们就使用简单的前缀，而没有采用匈牙利记法。我们觉得，如果变量命名合理，那么除了前缀应该不需要太多额外的上下文信息。在我们看来，一个数据成员命名为 mName，这就已经很清楚了。

好名字可以传达有关用途的信息，而不会使代码变得不可读。

## 7.5 合理地使用语言特性

C++ 语言允许做各种各样读起来很费解的事情。请看下面这行古怪的代码：

```
i++ + ++i;
```

由于 C++ 语言提供了强大的功能，重点是应该考虑如何正确地使用这些语言特性，从而得到好的风格，而不是糟糕的风格。

### 7.5.1 使用常量

不好的代码中通常充斥着“魔法数字”。在某个函数中，代码中会除 24。为什么是 24？是因为一天

有 24 小时吗？还是因为 New Brunswick 店的奶酪平均售价 \$ 24 呢？C++ 语言允许提供常量，为不会改变的值指定一个符号名，如可以为 24 指定一个符号常量。

```
const int kAveragePriceOfCheeseInNewBrunswick = 24;
```

### 7.5.2 利用 const 变量

C++ 中的 const 关键字实际上表示的是“不改变这个变量”，这种语法与其说有利于程序，不如说是要为程序员提供帮助。适当地使用 const，对编程风格的影响更大，而不是编程的正确性。有一些有经验的 C++ 程序员肯定认为没有理由使用 const，而且觉得不使用 const 也没有对自己的职业生涯带来任何负面影响。与 C++ 中的很多方面一样，const 的存在，更多地是为了帮助程序员，而不是程序。你应该使用 const，而且要正确地使用。const 的利与弊将在第 12 章介绍。以下是一个函数的原型，这个函数告诉它的调用者，它不会对传入的 C 风格字符串做任何改变。

```
void wontChangeString (const char* inString);
```

### 7.5.3 使用引用而不是指针

一般地，C++ 程序员先学的是 C。如果你也是这样，可能会发现引用并没有为 C++ 语言增加任何新的功能。它们只是对指针所提供的功能引入了一种新的语法。在 C 中，指针是惟一的传引用（按引用传递）机制，而且指针也确实很好地使用了许多年。在某些情况下，仍然需要指针，不过，在许多情况下，都可以转而使用引用。

使用引用而不是指针有许多优点。首先，引用比指针更安全，因为引用不会直接处理内存地址，而且不会为 NULL。其次，在风格上，引用也比指针更有优越性，因为引用使用的是栈变量的语法，这就避免了诸如 \* 和 & 等符号。引用也更易于使用，因此应当毫不迟疑地把采用引用纳入到你的编程风格当中。

引用还有一个优点，利用引用可以明确内存的所有权。如果你在编写一个方法，另一个程序员向你传递了一个对象的引用，显然你可以读取和修改这个对象，但是没有办法释放这个对象的内存。如果你传递的是一个指针，就没有这么清楚了。你需要删除对象来清空内存吗？或者这个工作是不是要由调用者来做？你的小组应当明确，采用不同的变量传递技术时，相应的变量内存将由谁所有。对此有一种简单的方法，即大家协商好，如果代码得到一个指针，就由代码拥有相应内存，而且要做必要的清除工作。所有其他变量要作为引用或副本传递（译者注：即你没有相应的内存所有权）。

以下函数原型清楚地指出参数会改变，不过，由于它是一个引用，所以不能释放它的内存。

```
void changeMe (ChessBoard& outBoard);
```

### 7.5.4 使用定制异常

C++ 中很容易忽略异常。在 C++ 语言中，没有哪个语法要求必须处理异常，通过传统方法（如返回 NULL，或者设置一个错误标志）也可以很容易地编写出容错的程序。

不过，异常为错误处理提供了一种功能更为丰富的机制，而利用定制异常，还可以适当地对此机制进行“剪裁”，来满足自己的需要。例如，对应 Web 浏览器的定制异常类型可能包括以下属性：包含错误的网页，错误发生时的网络状态，以及其他一些上下文信息。

第 15 章将对 C++ 中的异常做全面的介绍。

语言特性的存在是为了帮助程序员。要理解并充分利用会带来好的编程风格的特性。

## 7.6 格式化

许多编程小组都是因为对代码格式存在争议而导致分崩离析，友谊破裂。本书作者之一上大学的时候，曾经对一个if语句（用以判断是否一切正常）中空格的的使用与一位同事发生过激烈的争吵。如果你的组织已经为代码格式建立了标准，你应该很感幸运。尽管你可能并不喜欢这些标准，但起码不必为代码格式而与人大动干戈。如果你的小组中每个人都按自己的方式编写代码，那么就需要你有相当大的承受力才行。可以看到，有些实践做法只是各人口味不同而已（倒还能接受），但是还有一些做法则会产生很大分歧，导致小组很难协作。

### 7.6.1 有关大括号对齐的争论

也许争论最多的就是应该把界定代码块的大括号放在哪里。对于大括号的使用有许多不同的风格。在这本书中，我们会把大括号放在起始语句的同一行上，但对函数名、类名或方法名则除外。以下代码就显示了这种风格（本书都会采用这种风格）。

```
void someFunction()
{
    if (condition()) {
        cout << "condition was true" << endl;
    } else {
        cout << "condition was false" << endl;
    }
}
```

采用这种风格，可以在垂直方向上节省空间，而且仍能通过代码的缩进显示出代码块。有些程序员可能会提出异议，认为在实际的代码编写中节省垂直方向上的空间意义不大（特别是按代码行数计算报酬时，这样还有坏处！）。以下显示了一种更详细的风格。

```
void someFunction()
{
    if (condition())
    {
        cout << "condition was true" << endl;
    }
    else
    {
        cout << "condition was false" << endl;
    }
}
```

有些程序员还会在水平方向上大量使用空格，这就会得到如下例所示的代码。

```
void someFunction()
{
    if (condition())
    {
        cout << "condition was true" << endl;
    }
    else
    {
        cout << "condition was false" << endl;
    }
}
```

当然，我们并不会特别推荐哪一种特定的风格，因为任何一种风格都有不少支持者，我们不想得罪任何人。

在选择表示代码块的风格时，重要的是能否只通过查看代码，就清楚地看出哪个代码块落在哪个条件之下。

### 7.6.2 考虑空格和小括号

对于各个代码行的格式，同样也有不同的意见。我们还是不推崇哪一种特定的方法，不过在此将介绍几种可能会遇到的风格。

在这本书中，我们会在所有关键字后面使用一个空格，并使用小括号来明确操作的顺序。如下所示。

```
if (i == 2) {  
    j = i + (k / m);  
}
```

还有一种做法，从形式上看，在此将 if 看作是一个函数，关键字和左括号之间没有空格。另外，在这个 if 语句内部，并没有使用以上用来明确操作顺序的小括号，因为在此不存在语义关联（译者注：可以理解为不存在操作执行顺序的二义性），如下所示。

```
if(i == 2) {  
    j = i + k / m;  
}
```

二者的差别微乎其微，究竟哪一个更好，留给读者自己来判断，不过，必须指出，if 绝非一个函数。

### 7.6.3 空格和制表符

空格和制表符的使用并非只是有关风格的选择。如果你的小组未能对空格和制表符的约定达成一致意见，程序员联合工作时就会出现一些重大的问题。最显著的问题是，如果 Alice 使用 4 个空格来完成代码缩进，而 Bob 使用 5 个制表符来达到目的；在处理同一个文件时，他们都无法正确地显示代码。还有一个更糟糕的问题，如果 Bob 对代码重新格式化，从而全部使用制表符，而与此同时 Alice 编辑了这段代码，那么许多源码控制系统都无法将 Alice 所做的修改合并在内。

大多数（但并非全部）编辑器都为空格和制表符提供了可配置的设置。有些环境甚至还会在读入代码时调整代码的格式，或者即使原先编辑时使用的是 tab 键，也总是使用空格（而不用制表符）。如果你有一个灵活的环境，就更有希望顺利地处理其他人的代码。只要记住一点，制表符和空格是不同的，因为制表符可以是任意长度，而空格只是一个空格而已。出于这个原因，建议你使用一种总是将制表符解释为 4 个空格的编辑器。

## 7.7 风格方面的难题

许多程序员在开始一个新项目时，心里都默念：这一次希望一切顺利。只要有变量或参数不会改变，就会标记为 const。所有变量都有清晰、简洁、可读的名字。每一位开发人员都会把左大括号放在适当的位置，而且会采用标准文本编辑器，并遵循编辑器对制表符和空格的约定。

出于多种原因，风格很难保持这么高度的一致性。对于 const，有时程序员根本不知道如何来使用。你肯定会遇到一些老的代码或库函数，其中完全不知道使用 const。好的程序员会使用 const\_cast 临时“挂起”变量的 const 属性，而没经验的程序员可能会从调用函数“解放”const 属性，这样又会导致程序

没有使用 `const`。

在另外一些情况下，风格标准化可能与程序员自己的个人口味不相符。也许你的小组的特有文化氛围使得执行严格的风格原则并不实际。在这种情况下，可能必须确定需要对哪些方面标准化（如变量名和制表符），而哪些方面可以放心地留给个人来决定（如空格和注释风格）。你甚至还可以得到或编写一些脚本，从而自动地修正风格方面的“bug”，或者在指出代码错误的同时还指出风格方面存在的问题。

## 7.8 小结

C++ 语言提供了许多风格或样式方面的工具，但没有提供正式的指导原则来说明如何使用这些工具。幸运的是，评判任何风格约定的好坏时，标准都是一样的，即要衡量其采用的广泛程度如何，以及这种风格对提高代码的可读性有多大的好处。在参与团队开发时，讨论使用何种语言和工具时就应当尽早地提出有关风格的问题。

有关风格最重要的一点是，要认识到这是程序设计中的一个重要方面。在把代码交给别人之前，先检查一下代码的风格。另外应当从所用代码（别人所写）中发现好的风格，并采用你和你的组织认为有用的约定。

## 第 8 章 掌握类和对象

作为一种面向对象语言，C++ 提供了使用对象（object）和编写对象定义的“工具”，这称为类（class）。当然也可以用 C++ 编写没有类和对象的程序，不过这样一来，就无法利用到 C++ 语言中最基本、最有用的方面。如果编写一个没有类的 C++ 程序，就像是到了巴黎不吃法国餐而吃麦当劳一样！

为了有效地使用类和对象，必须理解类和对象的语法和功能。第 1 章回顾了类定义的基本语法。第 3 章介绍了用 C++ 设计程序的面向对象方法，并对类和对象提供了一些特定的设计策略。本章将介绍使用类和对象所涉及的一些基本概念，包括编写类定义、定义方法、使用栈和堆中的对象、编写构造函数、默认构造函数、编译器生成的构造函数、构造函数中的初始化列表、复制构造函数、析构函数以及赋值操作符。你可能已经对类和对象很了解，即便如此，也应当通读本章，因为其中包含了许多你可能并不熟悉的“内幕”信息。

### 8.1 电子表格示例

本章和下一章会提供一个可实际运行的例子，这是一个简单的电子表格应用。电子表格是一个由“单元格”（cell）组成的二维表格，每个单元格包含一个数字或字符串。如 Microsoft Excel 之类的专业电子表格应用还提供了完成数学运算的功能，例如可以计算一组单元格中值的总和。这两章中的电子表格例子并不打算在市场上抢占 Microsoft 的份额，这个例子只是用来说明有关类和对象的问题。

电子表格应用使用了两个基本的类：Spreadsheet 和 SpreadsheetCell。每个 Spreadsheet 对象都包含多个 SpreadsheetCell 对象。另外，一个 SpreadsheetApplication 类管理多个 Spreadsheet。本章重点介绍 SpreadsheetCell。第 9 章将开发 Spreadsheet 和 SpreadsheetApplication 类。

这一章提供了多个不同版本的 SpreadsheetCell 类，从而逐步地介绍各个相关概念。因此，本章中对类实现的各种尝试并不一定是编写类时完成各个方面的“最佳”途径。具体地，开始给出的例子就忽略了一些重要的特性（而一般都应包括这些特性才对），这只是因为当时还没有介绍到有关特性，所以才未使用。引言中给出了源代码的下载地址，你可以从中下载这个类的最后版本。

### 8.2 编写类

在编写一个类时，要指定应用于该类对象的行为（或方法，method）和每个对象所包含的属性（或数据成员，data member）。

编写类包括两个方面：定义类本身以及定义类的方法。

#### 8.2.1 类定义

以下是对简单的 SpreadsheetCell 类的第一次尝试，在此，每个单元格只存储一个数字。



```
// SpreadsheetCell.h
class SpreadsheetCell
{
public:
    void setValue(double inValue);
    double getValue();

protected:
    double mValue;
};
```

如第1章所述，每个类定义的前面都有关键字 `class`，接下来是类名。类定义在 C++ 中是一条语句 (statement)，因此必须以分号结束。如果类定义的后面没有加分号，编译器就可能提示出许多错误，其中大多数错误都与实际问题 (少了分号) 毫不相干。

类定义通常放在一个名为 `ClassName.h` 的文件中 (译者注：这里的 `ClassName` 即为实际的类名)。

### 方法和成员

以下两行代码看上去就像是函数原型，它们声明了这个类所支持的方法 (译者注：这本书中提到“函数”时，一般指不是类中成员方法的独立函数，而提到“方法”时，则指类中的成员方法)。

```
void setValue (double inValue);
double getValue();
```

以下这行代码看上去就如同一个变量声明，它声明了这个类的数据成员：

```
double mValue;
```

每个对象都有自己的 `mValue` 变量。不过，方法的实现则由所有对象共享。类可以包含任意多个方法和成员 (译者注：这里的成员是指数据成员或成员变量，但一般文献中往往把成员变量和成员方法统称为成员)。成员不能与方法同名。

### 访问控制

类中的每个方法和成员都有一个访问限定符 (access specifier) (译者注：一般也称为修饰符，modifier)，并受其制约，访问限定符可以是以下三者之一：`public`、`protected` 或 `private`。访问限定符应用于其后的所有方法和成员声明，直到遇到下一个访问限定符为止。在 `SpreadsheetCell` 类中，`setValue()` 和 `getValue()` 方法的访问限定符为 `public`，`mValue` 成员的访问限定符为 `protected`。

```
public:
    void setValue(double inValue);
    double getValue();
protected:
    double mValue;
```

类的默认访问限定符是 `private`，第一个访问限定符之前的所有方法和成员声明实际上都指定为 `private` 访问。例如，把 `public` 访问限定符移到 `setValue()` 方法声明下面时，就会置对 `setValue()` 的访问为 `private`，而不是 `public`。

```
class SpreadsheetCell
{
    void setValue(double inValue); // now has private access
public:
    double getValue();
```

```
protected:
    double mValue;
};
```

在 C++ 中, struct (结构) 也可以像类一样有方法。实际上, 结构和类之间惟一的差别就在于结构的默认访问限定符为 public, 而类的默认访问限定符为 private。

表 8-1 对三种访问限定符的含义做了总结。

表 8-1

访问限定符	含 义	何时使用
public	任何代码都可以调用一个 public 方法, 或访问一个对象的 public 成员	希望客户使用的行为 (方法) 可以置为 public private 和 protected 数据成员的存取方法要置为 public
protected	当前类的任何方法都可以调用 protected 方法, 以及访问 protected 成员。子类 (subclass) 的方法也可以调用 protected 方法, 或访问一个对象的 protected 成员。子类的介绍见第 10 章	不希望客户使用的“辅助”方法可以置为 protected 大多数数据成员都置为 protected
private	只有此类的方法才能调用 private 方法, 以及访问 private 成员。子类中的方法不能访问 private 方法或成员	只有希望限制子类对方法或成员的访问时, 才使用 private

访问限定符作用在类层次上, 而不是对象层次, 因此一个类的方法可以访问该类任何对象的 protected 或 private 方法及成员。

### 声明顺序

可以按任何顺序声明方法、成员和访问控制限定符, C++ 对此未做任何限制, 如没有要求方法必须在成员之前, 也没有要求 public 必须在 private 之前。另外, 访问限定符可以重复出现。例如, SpreadsheetCell 定义可能如下所示:

```
class SpreadsheetCell
{
public:
    void setValue(double inValue);

protected:
    double mValue;

public:
    double getValue();
};
```

不过, 为清楚起见, 最好是分组放置 public、protected 和 private 声明, 即把访问限定符相同的声明归在一起, 另外, 这些声明中的方法和成员也分别归组。在本书中, 我们将如下安排类中 (方法及成员) 定义和访问限定符的顺序:

```
class ClassName
{
public:
    // Method declarations
    // Member declarations
```

```
protected:
    // Method declarations
    // Member declarations

private:
    // Method declarations
    // Member declarations
};
```

### 8.2.2 定义方法

基于前面对 SpreadsheetCell 类的定义，完全可以创建这个类的对象了。不过，如果你尝试调用 setValue() 或 getValue() 方法，连接器就会提示错误，指出这些方法未定义。这是因为，类定义只是指定了方法的原型，并未定义其实现。编写独立函数（也就是说，不是类中的方法）时，不仅要编写它的原型，还要提供一个定义，与此类似，对于类中的方法也必须提供一个原型和一个定义。注意，类定义必须在方法定义之前。通常类定义会放在一个头文件中，而方法定义放在一个源文件中，这个源文件要通过 includes 宏包含该头文件。以下是 SpreadsheetCell 类中两个方法的定义：

```
// SpreadsheetCell.cpp
#include "SpreadsheetCell.h"

void SpreadsheetCell::setValue(double inValue)
{
    mValue = inValue;
}

double SpreadsheetCell::getValue()
{
    return (mValue);
}
```

注意每个方法名前面都有该类的类名及两个冒号：

```
void SpreadsheetCell:: setValue (double value)
```

:: 称为作用域解析操作符 (scope resolution operator)。在这里，上述语法会告诉编译器后面的 setValue() 方法定义是 SpreadsheetCell 类的一部分。还要注意，定义方法时，没有重复加上访问限定符。

#### 访问数据成员

类的大多数方法（如 setValue() 和 getValue()）总是在该类的一个特定对象上执行（但静态方法有所例外，后面将讨论有关内容）。在方法体内部，可以访问该对象相应类的所有数据成员。在前面的 setValue() 定义中，下面这行代码会修改调用此方法的任何对象中的 mValue 变量：

```
mValue = inValue;
```

如果两个不同的对象调用了 setValue()，同样这行代码（分别为每个对象执行一次）会改变两个不同对象中的变量。

#### 调用其他方法

可以从另一个方法内部调用一个类的方法。例如，考虑对 SpreadsheetCell 类的一个扩展。实际的电子表格应用不仅允许单元格中有数字，还允许有文本数据。如果想把一个文本单元格解释为一个数字，

电子表格会尝试将文本转换为一个数字。如果文本不表示合法的数，单元格的值会被忽略。在这个程序中，遇到非数字的字符串时，会生成一个值为 0 的单元格。以下是对类定义的第一次调整，从而建立一个支持文本数据的 SpreadsheetCell。

```
#include <string>
using std::string;

class SpreadsheetCell
{
public:
    void setValue(double inValue);
    double getValue();
    void setString(string inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(string inString);

    double mValue;
    string mString;
};
```

这个版本的类不仅存储数据的数字表示，还存储文本表示。如果客户将数据设置为一个 string，它会转换为一个 double，如果客户将数据设置为一个 double，double 会转换为一个 string。如果文本不是一个合法的数字，double 值则为 0。这个类定义显示了两个新的方法来设置和获取单元格的文本表示，还有两个新的 protected 辅助方法，用以将一个 double 转换为一个 string，以及将一个 string 转换为一个 double。这些辅助方法使用了字符串流，有关内容将在第 14 章详细介绍。以下是所有方法的实现。

```
#include "SpreadsheetCell.h"
```

```
#include <iostream>
#include <sstream>
using namespace std;
```

```
void SpreadsheetCell::setValue(double inValue)
{
    mValue = inValue;
    mString = doubleToString(mValue);
}
```

```
double SpreadsheetCell::getValue()
{
    return (mValue);
}
```

```
void SpreadsheetCell::setString(string inString)
{
    mString = inString;
    mValue = stringToDouble(mString);
}
```

```
string SpreadsheetCell::getString()
{

```

```

        return (mString);
    }

    string SpreadsheetCell::doubleToString(double inValue)
    {
        ostringstream ostr;

        ostr << inValue;
        return (ostr.str());
    }

    double SpreadsheetCell::stringToDouble(string inString)
    {
        double temp;

        istringstream istr(inString);

        istr >> temp;
        if (istr.fail() || !istr.eof()) {
            return (0);
        }
        return (temp);
    }

```

注意，每个设置方法都调用了辅助方法来完成一个转换。基于这种技术，mValue 和 mString 总是有效的。

### this 指针

每个常规的方法调用都会传递调用该方法的对象的一个指针，将其作为“隐藏”的第一个参数，名为 this。可以使用这个指针来访问数据成员或调用方法，还可以将其传递给其他方法或函数。有时这个指针还有助于消除名字的歧义。例如，可能如下定义了 SpreadsheetCell 类，使得 setValue() 方法取一个名为 mValue 的参数（而不是 inValue）。在这种情况下，setValue() 可能如下所示：

```

void SpreadsheetCell::setValue(double mValue)
{
    mValue = mValue; // Ambiguous!
    mString = doubleToString(mValue);
}

```

第一行代码就很让人摸不着头脑。你指的到底是哪一个 mValue，是作为参数传递的 mValue，还是作为对象成员的 mValue？为了消除名字的这种歧义，可以使用 this 指针：

```

void SpreadsheetCell::setValue(double mValue)
{
    this->mValue = mValue;
    mString = doubleToString(this->mValue);
}

```

不过，如果使用第7章所述的命名约定，就不会遇到这种命名冲突了。

还可以使用 this 指针从对象的一个方法中调用一个取对象指针作为参数的函数或方法。例如，假设编写了一个独立函数（而非类中的方法）printCell()，如下所示：



```
void printCell(SpreadsheetCell* inCellp)
{
    cout << inCellp->getString() << endl;
}
```

如果想从 setValue() 方法调用 printCell(), 必须将 this 作为参数传递, 从而向 printCell() 提供一个 SpreadsheetCell 对象 (即调用 setValue() 的对象, setValue() 就在此对象上操作) 的指针:

```
void SpreadsheetCell::setValue(double mValue)
{
    this->mValue = mValue;
    mString = doubleToString(this->mValue);
    printCell(this);
}
```

### 8.2.3 使用对象

前面的类定义指出, 一个 SpreadsheetCell 包括两个成员变量, 4 个公共方法, 以及两个保护方法。不过, 这个类定义并没有真正创建任何 SpreadsheetCell, 它只是指出了单元格的格式。从这个意义上讲, 类就类似于建筑蓝图。蓝图指定了一个房子应该是什么样, 但是画蓝图并不是真正盖房子。房子必须过后根据蓝图来构造。

类似地, 在 C++ 中, 可以根据 SpreadsheetCell 类定义来构造一个 SpreadsheetCell “对象”, 即声明一个类型为 SpreadsheetCell 的变量。建筑工人基于一个给定的蓝图可以盖多个房子, 与此类似, 程序员也可以根据一个 SpreadsheetCell 类创建多个 SpreadsheetCell 对象。有两种创建和使用对象的方法: 在栈上创建和在堆上创建。

#### 栈上的对象

以下是在栈上创建和使用 SpreadsheetCell 对象的一些代码:

```
SpreadsheetCell myCell, anotherCell;
myCell.setValue(6);
anotherCell.setValue(myCell.getValue());

cout << "cell 1: " << myCell.getValue() << endl;
cout << "cell 2: " << anotherCell.getValue() << endl;
```

可以像声明简单变量一样创建对象, 只不过这里的变量类型是类名。以上代码行 (如 myCell.setValue(6);) 中的, 称为“点”操作符, 利用点操作符可以调用对象上的方法。如果对象中存在公共数据成员, 那么还可以利用点操作符访问这些公共数据成员。

这个程序的输出是:

```
cell 1: 6
cell 2: 6
```

#### 堆上的对象

还可以使用 new 动态分配对象:

```
SpreadsheetCell* myCellp = new SpreadsheetCell();

myCellp->setValue(3.7);
cout << "cell 1: " << myCellp->getValue() <<
    " " << myCellp->getString() << endl;
delete myCellp;
```

在堆上创建一个对象时,要通过“箭头”操作符(→)来调用方法,以及访问成员。箭头结合了两个作用,一方面解除引用(\*),另一方面完成方法或成员的访问(.)。也可以使用这两个操作符(\*和.),但是这样做的风格不是太好:

```
SpreadsheetCell* myCellp = new SpreadsheetCell();
```

```
(*myCellp).setValue(3.7);  
cout << "cell 1: " << (*myCellp).getValue() <<  
    * << (*myCellp).getString() << endl;  
delete myCellp;
```

就像必须释放在堆上分配的其他内存一样,必须通过对对象调用 delete 来释放堆上为对象所分配的内存。

如果用 new 来分配一个对象,用完该对象时就要用 delete 来释放。

## 8.3 对象生命期

对象生命期涉及三个活动:创建、撤销和赋值。每个对象都会创建,但是并非所有对象都能碰到另外两个“生命事件”。一定要理解对象如何以及何时创建、撤销和赋值,还要了解如何定制这些行为,这些非常重要。

### 8.3.1 对象创建

如果是栈上的对象,在声明对象时就会创建对象,或者,对于堆上的对象,在用 new 或 new[] 显式为对象分配空间时,就会创建对象。

在声明变量时给出变量初始值往往很有帮助。例如,通常应当如下将整型变量初始化为 0:

```
int x = 0, y = 0;
```

类似地,也应当为对象提供初始值。可以声明并编写一个特殊的方法来提供这个功能,这个方法称为构造函数 (constructor),在构造函数中,可以完成对象的初始化工作。只要创建一个对象,就会执行它的一个构造函数。

C++ 程序员通常把构造函数称做“ctor”。

#### 编写构造函数

下面是为 SpreadsheetCell 类增加的第一个构造函数:

```
class SpreadsheetCell  
{  
public:  
    SpreadsheetCell(double initialValue);  
    void setValue(double inValue);  
    double getValue();  
    void setString(string inString);  
    string getString();  
  
protected:  
    string doubleToString(double inValue);  
    double stringToDouble(string inString);  
  
    double mValue;  
    string mString;  
};
```



注意，构造函数与类同名，而且不能有返回类型。构造函数一定要保证这两点。就像必须为一般的方法提供实现一样，也要为构造函数提供实现：

```
SpreadsheetCell::SpreadsheetCell(double initialValue)
{
    setValue(initialValue);
}
```

SpreadsheetCell 构造函数是 SpreadsheetCell 类的一个方法，因此 C++ 要求方法名前面要有常规的 SpreadsheetCell:: 作用域解析短语。方法名本身也是 SpreadsheetCell，所以最后的代码看上去就有些滑稽 (SpreadsheetCell:: SpreadsheetCell.)。这个实现只是调用一个 setValue()，从而设置数字和文本表示。

### 使用构造函数

使用构造函数创建一个对象并初始化它的值。可以利用基于栈和基于堆的分配来使用构造函数。

#### 在栈上使用构造函数

在栈上分配一个 SpreadsheetCell 对象时，可以如下使用构造函数：

```
SpreadsheetCell myCell(5), anotherCell(4);

cout << "cell 1: " << myCell.getValue() << endl;
cout << "cell 2: " << anotherCell.getValue() << endl;
```

注意，你并没有显式地调用 SpreadsheetCell 构造函数。例如，不要写出以下代码：

```
SpreadsheetCell myCell. SpreadsheetCell (5); // WILL NOT COMPILE!
```

类似地，也不能以后再调用构造函数。下面的做法也是不正确的：

```
SpreadsheetCell myCell;
myCell.SpreadsheetCell(5); // WILL NOT COMPILE!
```

重申一次，在栈上使用构造函数的惟一正确的方法如下所示：

```
SpreadsheetCell myCell (5);
```

#### 在堆上使用构造函数

动态分配一个 SpreadsheetCell 对象时，要如下使用构造函数：

```
SpreadsheetCell *myCellp = new SpreadsheetCell(5);
SpreadsheetCell *anotherCellp;
anotherCellp = new SpreadsheetCell(4);
delete anotherCellp;
```

注意，你可以声明一个 SpreadsheetCell 对象的指针，但不必立即调用构造函数，这与栈上的对象有所不同，在栈上，只要声明对象就会调用构造函数。

与以往一样，如果用 new 动态分配了对象，一定要记住要对这些对象调用 delete！

### 提供多个构造函数

在一个类中可以提供不只一个构造函数。所有构造函数都同名（即均为类名），不过不同的构造函数必须有不同数目的参数，或者参数类型有所不同。

在 SpreadsheetCell 类中，如果有两个构造函数，一个取初始 double 值为参数，另一个取初始 string 值为参数，这会很有帮助。以下是加了第二个构造函数的类定义：

```
class SpreadsheetCell
{
public:
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(string initialValue);
    void setValue(double inValue);
    double getValue();
    void setString(string inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(string inString);

    double mValue;
    string mString;
};
```

以下是第二个构造函数的实现：

```
SpreadsheetCell::SpreadsheetCell(string initialValue)
{
    setString(initialValue);
}
```

以下代码中使用了这两个不同的构造函数：

```
SpreadsheetCell aThirdCell("test"); // Uses string-arg ctor
SpreadsheetCell aFourthCell(4.4);    // Uses double-arg ctor
SpreadsheetCell* aThirdCellp = new SpreadsheetCell("4.4"); // string-arg ctor
cout << "aThirdCell: " << aThirdCell.getValue() << endl;
cout << "aFourthCell: " << aFourthCell.getValue() << endl;
cout << "aThirdCellp: " << aThirdCellp->getValue() << endl;
delete aThirdCellp;
```

如果有多个构造函数，往往希望利用一个构造函数来实现另一个构造函数。例如，可能想从 string 版本的构造函数（即取 string 参数的构造函数）调用 double 版本的构造函数，如下所示：

```
SpreadsheetCell::SpreadsheetCell(string initialValue)
{
    SpreadsheetCell(stringToDouble(initialValue));
}
```

看上去好像是合理的。毕竟，正常的类方法确实可以在其他方法中调用。这段代码可以编译、连接并运行，不过结果却并非想像的那样。显式地调用 SpreadsheetCell 构造函数实际上会创建一个类型为 SpreadsheetCell 的新的临时匿名对象。它不会调用你想初始化的那个对象的构造函数。

不要尝试从类的一个构造函数调用另一个构造函数。

### 默认构造函数

默认构造函数（default constructor）是一个无参数的构造函数。这也称为 0 参数构造函数（0-argument constructor）。利用默认构造函数，可以为数据成员指定合理的初始值，即使客户未做指定也不妨碍。

以下是带有一个默认构造函数的 SpreadsheetCell 类定义中的一部分：

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(string initialValue);
    // Remainder of the class definition omitted for brevity
};
```

以下是对默认构造函数的第一个实现：

```
SpreadsheetCell::SpreadsheetCell()
{
    mValue = 0;
    mString = "";
}
```

可以如下在栈上使用默认构造函数：

```
SpreadsheetCell myCell;
myCell.setValue(6);

cout << "cell 1: " << myCell.getValue() << endl;
```

前面的代码会创建一个名为 myCell 的新 SpreadsheetCell，设置其值，并打印出它的值。对于基于栈的对象，不同于其他构造函数，不能采用函数调用的语法来调用默认构造函数。如果非要采用其他构造函数的语法，可能会这样来调用默认构造函数：

```
SpreadsheetCell myCell(); // WRONG, but will compile.
myCell.setValue(6);       // However, this line will not compile.

cout << "cell 1: " << myCell.getValue() << endl;
```

遗憾的是，调用默认构造函数的那一行会通过编译，但是后面的一行将无法通过编译。这里的问题是：编译器会认为第一行实际上是一个函数的函数声明，此函数名为 myCell，不带任何参数，并返回一个 SpreadsheetCell 对象。编译到第二行时，会认为你想将一个函数名用作作为一个对象。

在栈上创建对象时，要去掉默认构造函数的小括号。

不过，对于基于堆的对象分配，使用默认构造函数时，则必须采用函数调用语法：

```
SpreadsheetCell* myCellp = new SpreadsheetCell(); // Note the function-call syntax
```

在使用默认构造函数时，对于基于堆和基于栈的对象分配，C++ 要求采用不同的语法，不要浪费太多时间去考虑为什么会这样。人们为什么会这么热衷于学习 C++，就是因为它有许多有趣的地方，这正是其中之一。

#### 编译器生成的默认构造函数

如果类没有提供一个默认构造函数，如果不指定参数就无法创建该类的对象。例如，假设有以下 SpreadsheetCell 类定义：

```
class SpreadsheetCell
{
public:
```

```
SpreadsheetCell(double initialValue); // No default constructor
SpreadsheetCell(string initialValue);
void setValue(double inValue);
double getValue();
void setString(string inString);
string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(string inString);

    double mValue;
    string mString;
};
```

基于上述定义，以下代码将不能编译：

```
SpreadsheetCell myCell;
myCell.setValue(6);
```

不过，这两行代码以前是可以的！这里出了什么错吗？错倒没有出。因为没有声明默认构造函数，所以不指定参数的话当然就无法构造一个对象。

真正的问题是为什么原来这个代码能正常工作（原来也没有指定默认构造函数呀！）。原因在于，如果你没有指定任何构造函数，编译器就会写出一个不带任何参数的构造函数。这个编译器生成的默认构造函数会调用类中所有对象成员的默认构造函数，但是不会对诸如 int 和 double 等基本类型的成员进行初始化。这样一来，你就能创建该类的对象了。然而，如果你声明了一个默认构造函数，或者任何其他构造函数，编译器就不再生成默认构造函数了。

默认构造函数与 0 参数（无参数）构造函数是同一个东西。“默认构造函数”一词并不只表示未声明任何构造函数时会自动生成的构造函数。

### 什么时候需要默认构造函数

可以考虑对象数组。创建一个对象数组完成了两个任务：会为所有对象分配连续的空间，而且会对每个对象调用默认构造函数。数组创建代码不能调用其他的构造函数，对此 C++ 没有提供任何语法。例如，如果你没有为 SpreadsheetCell 类定义一个默认构造函数，以下代码将不能编译：

```
SpreadsheetCell cells[3]; // FAILS compilation without a default constructor
SpreadsheetCell* myCellp = new SpreadsheetCell[10]; // Also FAILS
```

对于基于栈的数组，也可以使用初始化方法（initializer）绕过这个限制，如下所示：

```
SpreadsheetCell cells[3] = {SpreadsheetCell(0), SpreadsheetCell(23),
    SpreadsheetCell(41)};
```

不过，如果想创建某个类的对象数组，最好还是要确保该类有一个默认构造函数。

如果想在其他类中创建某个类的对象，默认构造函数也很有用，请参见下一节“初始化列表”中的说明。

最后一点，如果类作为一个继承层次体系的基类时，有默认构造函数会很方便。在这种情况下，子类可以很方便地通过超类的默认构造函数来初始化超类。第 10 章将更详细地讨论这个问题。

### 初始化列表

C++ 还提供了另外一种方法，以便在构造函数中初始化数据成员，这称为初始化列表（initializer

list)。以下是重写后的 0 参数 SpreadsheetCell 构造函数，这里使用了初始化列表语法：

```
SpreadsheetCell::SpreadsheetCell() : mValue(0), mString("")
{
}
```

可以看到，初始化列表位于构造函数参数表和构造函数体的开始大括号之间。初始化列表以一个冒号开头，并以逗号作为分隔符。列表中的每个元素分别对一个数据成员初始化，可以使用函数记法，也可以调用一个超类构造函数（见第 10 章）。

利用一个初始化列表来完成数据成员的初始化，这与在构造函数体本身初始化数据成员有所不同。C++ 创建一个对象时，在调用构造函数之前必须先创建该对象的所有数据成员。作为创建这些数据成员的一部分，如果有的数据成员本身是对象，则必须调用这些对象的构造函数。在构造函数体中为一个对象赋值时，并没有真正构造该对象，而只是修改了它的值。初始化列表则允许在创建数据成员的同时为之提供初始值，这比以后再赋值效率更高。有意思的是，string 的默认初始化就会赋为空串，所以上例中显式地将 mString 初始化为空串有些多余。

初始化列表允许在创建数据成员的同时完成数据成员的初始化。

即使你不太在意效率，也可能想使用初始化列表，因为这样看起来更“简洁”。有些程序员更喜欢在构造函数体中赋初值的做法，这是一种更常用的语法。不过，有一些数据类型必须在初始化列表中进行这初始化。表 8-2 对此做了一个总结。

表 8-2

数据类型	解 释
const 数据成员	创建一个 const 变量后再对其赋值是不合法的。必须在创建的时候提供值
引用数据成员	如果不引用具体的东西，引用是无法独立存在的
没有默认构造函数的对象数据成员	C++ 会使用默认构造函数来初始化成员对象。如果成员对象没有默认构造函数，就无法初始化
没有默认构造函数的超类	[见第 10 章的介绍]

关于初始化列表有一个重要的警告：数据成员会按其出现在类定义中的顺序得到初始化，而不是按出现在初始化列表中的顺序。例如，假设将 SpreadsheetCell 的 string 版本的构造函数重写为使用初始化列表，如下所示：

```
SpreadsheetCell::SpreadsheetCell(string initialValue) :
    mString(initialValue), mValue(stringToDouble(mString))    // INCORRECT ORDER!
{
}
```

这段代码可以通过编译（不过有些编译器会提出一个警告），但是程序不能正确地工作。你可能认为 mString 会比 mValue 先初始化，因为在初始化列表中 mString 列在前面。不过，C++ 可不是这么做的。SpreadsheetCell 类先声明了 mValue，然后才声明 mString：

```
class SpreadsheetCell
{
public:
    // Code omitted for brevity
```



```
protected:
    // Code omitted for brevity
    double mValue;
    string mString;
};
```

因此初始化列表会在 mString 之前先初始化 mValue。不过，初始化 mValue 的代码要使用 mString 的值，但此时 mString 还没有初始化呢！对于这种情况，解决办法是在初始化 mValue 时使用 initialValue 参数，而不使用 mString。还应当把初始化列表中的顺序换过来，以避免混淆：

```
SpreadsheetCell::SpreadsheetCell(string initialValue) :
    mValue(stringToDouble(initialValue)), mString(initialValue)
{
}
```

初始化列表会按类定义中数据成员声明的顺序进行初始化，而不按出现在初始化列表中的顺序来初始化数据成员。

### 复制构造函数

C++ 中有一个特殊的构造函数，称为复制构造函数 (copy constructor)，复制构造函数允许创建一个对象作为另一个对象的完全副本。如果你自己没有编写一个复制构造函数，C++ 会生成一个，它会根据源对象中相应的数据成员，在新对象中初始化各个数据成员。对于对象数据成员（译者注：即数据成员本身是对象），这里的初始化意味着会调用这些对象数据成员的复制构造函数。

以下是 SpreadsheetCell 类中复制构造函数的声明：

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(string initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    void setValue(double inValue);
    double getValue();
    void setString(string inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(string inString);

    double mValue;
    string mString;
};
```

这个复制构造函数取源对象的一个 const 引用作为参数。与其他构造函数一样，它也不返回任何值。在这个构造函数内部，应当从源对象复制所有数据字段。当然，从理论上讲，在构造函数中可以做任何事情，但是通常好的做法是循规蹈矩地将新对象初始化为原对象的一个副本。以下是 SpreadsheetCell 复制构造函数的一个示例实现：

```

SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell& src) :
    mValue(src.mValue), mString(src.mString)
{
}

```

要注意初始化列表的使用。在初始化列表中设置值和复制构造函数体中设置值有所区别，这会在以下有关赋值一节中讨论。

编译器生成的 SpreadsheetCell 复制构造函数与以上复制构造函数完全相同。因此，为简单起见，可以不要显式地复制构造函数，而依赖编译器生成的复制构造函数。第 10 章将介绍另外一些类型的类，对于这些类，编译器生成的复制构造函数就不能满足要求了。

### 什么时候调用复制构造函数

C++ 中向函数传递参数的默认语义是按值传递（传值）。这说明，函数或方法接收变量的一个副本，而不是变量本身。因此，只要向一个函数或方法传递对象，编译器就会调用新对象的复制构造函数来进行初始化。

例如，假设 SpreadsheetCell 类中 setString() 方法的定义如下所示：

```

void SpreadsheetCell::setString(string inString)
{
    mString = inString;
    mValue = stringToDouble(mString);
}

```

还应该记得，C++ string 实际上是一个类，而不是内置的简单类型。如果代码传递了一个字符串实参来调用 setString()，则会调用这个字符串的复制构造函数来初始化 inString 字符串参数。复制构造函数的实参就是你传入 setString() 的字符串。在以下例子中，会在 setString() 中执行 inString 对象的串复制构造函数，在此以 name 作为其参数。

```

SpreadsheetCell myCell;
string name = "heading one";

myCell.setString(name); // Copies name

```

setString() 方法完成时，会撤销 inString。由于它只是 name 的一个副本，name 会原封不动地保持不变。

如果从一个函数或方法返回一个对象，也会调用复制构造函数。在这种情况下，编译器会通过复制构造函数创建一个临时的匿名对象。第 17 章将更详细地讨论临时对象的影响。

### 显式地调用复制构造函数

还可以显式地使用复制构造函数。将一个对象构造为另一个对象的完全副本，这通常很有用。例如，你可能希望如下创建 SpreadsheetCell 对象的一个副本：

```

SpreadsheetCell myCell2(4);
SpreadsheetCell anotherCell(myCell2); // anotherCell now has the values of myCell2

```

### 按引用传递对象

在向函数和方法传递对象时，为了避免复制对象，可以将函数或方法声明为取该对象的引用作为参数。按引用传递对象通常比按值传递效率更高，因为这样只会复制对象的地址，而不会复制对象的整个



内容。另外，按引用传递还可以避免对象动态内存分配的相关问题，我们将在第9章讨论这些内容。

应当将对象按 const 引用传递，而不是按值传递。

按引用传递一个对象时，使用对象引用的函数或方法可以改变原来的对象。如果你只是为了提高效率而使用按引用传递，还应当将对象声明为 const，从而禁止这种可能性。以下是 SpreadsheetCell 的类定义，在此字符串对象按 const 引用传递。

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(const string& initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    void setValue(double inValue);
    double getValue();
    void setString(const string& inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(const string& inString);

    double mValue;
    string mString;
};
```

以下是 setString() 的实现。注意这里的方法体还是一样的，只是参数类型有所不同。

```
void SpreadsheetCell::setString(const string& inString)
{
    mString = inString;
    mValue = stringToDouble(mString);
}
```

SpreadsheetCell 中返回 string 的方法还是按值来返回对象。返回一个数据成员的引用有很大风险，因为只有当对象“存活”时引用才有效。一旦对象撤销，该对象的引用也将失效。不过，有时出于一些合理的理由也会返回数据成员的引用，在本章后面以及以后的章节中就会看到这种情况。

### 编译器生成的构造函数小结

编译器会为每一个类自动生成一个 0 参数构造函数和一个复制构造函数（译者注：但这是有条件的，见表 8-3）。不过，你自己定义的构造函数可以根据以下规则取代这些构造函数。

表 8-3

如果定义了……	……则编译器会生成……	……可以如下创建一个对象……	举 例
未定义构造函数	一个 0 参数构造函数 一个复制构造函数	不带参数 作为另一个对象的副本	SpreadsheetCell cell; SpreadsheetCell myCell (cell);
仅一个 0 参数构造函数	一个复制构造函数	不带参数 作为另一个对象的副本	SpreadsheetCell cell; SpreadsheetCell myCell (cell);
仅一个复制构造函数	无构造函数	理论上可以作为另一个对象的副本。实际中，无法创建任何对象	没有例子

(续)

如果定义了……	……则编译器会生成……	……可以如下创建一个对象……	举 例
仅一个单参数（非复制）构造函数或多参数构造函数	一个复制构造函数	带参数 作为另一个对象的副本	SpreadsheetCell cell(6); SpreadsheetCell myCell(cell);
一个 0 参数构造函数和一个单参数（非复制）构造函数或多参数构造函数	一个复制构造函数	不带参数 带参数 作为另一个对象的副本	SpreadsheetCell cell; SpreadsheetCell myCell(5); SpreadsheetCell anotherCell(cell);

注意，默认构造函数与复制构造函数之间不存在对称性。只要没有显式定义一个复制构造函数，编译器就会创建一个。另一方面，只要定义了任何构造函数，编译器则不再生成默认构造函数。

### 8.3.2 对象撤销

撤销一个对象时，会出现两个事件：一是将调用对象的撤销方法；二是要释放对象所占用的空间。析构函数提供了一个机会，可以在此完成对象的所有清除工作，如释放动态分配的内存，或关闭文件句柄等。如果没有声明析构函数，编译器会生成一个析构函数，递归地完成各个成员的撤销，从而可以删除对象。第 9 章有一节专门介绍动态内存分配，在那里你将了解到如何编写一个析构函数。

对于栈上的对象，如果出了作用域（out of scope），对象就会被撤销，这说明只要当前函数、方法或其他执行模块结束，就会撤销它在栈上创建的对象。换句话说，只要代码遇到一个结束大括号，当前大括号对（即开始大括号和结束大括号）之间如果在栈中创建了对象，那么所有这些对象都会被撤销。以下程序展示了这种行为。

```
int main(int argc, char** argv)
{
    SpreadsheetCell myCell(5);

    if (myCell.getValue() == 5) {
        SpreadsheetCell anotherCell(6);
    } // anotherCell is destroyed as this block ends.

    cout << "myCell: " << myCell.getValue() << endl;

    return (0);
} // myCell is destroyed as this block ends.
```

栈中的对象会按其声明（和构造）的逆序撤销。例如，在以下代码段中，myCell2 在 anotherCell2 之前分配，因此 anotherCell2 要先于 myCell2 被撤销（需要说明，在程序中的任何位置都可以用一个开始大括号作为一个新的代码块的开始）：

```
{
    SpreadsheetCell myCell2(4);
    SpreadsheetCell anotherCell2(5); // myCell2 constructed before anotherCell2
} // anotherCell2 destroyed before myCell2
```

如果对象的数据成员本身也是对象，同样会如此逆序撤销。应该记得，数据成员是按其在类中声明的顺序来初始化的。因此，遵照上述规则，对象按构造的逆序撤销，数据成员对象就会按其在类中声明的逆序撤销。

堆上分配的对象不会自动撤销。必须对对象指针调用 delete，从而调用析构函数，并释放内存。以下

程序展示了这种行为：

```
int main(int argc, char** argv)
{
    SpreadsheetCell* cellPtr1 = new SpreadsheetCell(5);
    SpreadsheetCell* cellPtr2 = new SpreadsheetCell(6);

    cout << "cellPtr1: " << cellPtr1->getValue() << endl;

    delete cellPtr1; // Destroys cellPtr1

    return (0);
} // cellPtr2 is NOT destroyed because delete was not called on it.
```

### 8.3.3 对象赋值

在C++中，如同将一个int的值赋给另一个int一样，也可以将一个对象的值赋给另一个对象。例如，以下代码会把myCell的值赋给anotherCell：

```
SpreadsheetCell myCell(5), anotherCell;
```

```
anotherCell = myCell;
```

你可能想说myCell被“复制”到anotherCell。不过，在C++世界中，“复制”只会在对象初始化时才会出现。如果一个对象已经有值，而这个值要被重写或覆盖，更准确的说法是“赋值”（assigned）。需要注意，C++为复制所提供的工具是复制构造函数。由于复制构造函数是一个构造函数，因此只能用于对象创建，而不能用于以后对对象的赋值。

因此，C++在每个类中提供了另外一个方法来完成赋值。这个方法称为赋值操作符（assignment operator）。具体名为operator=，因为这实际上是针对该类对=操作符的一个重载。在上述例子中，就调用了anotherCell的赋值操作符，在此以myCell作为参数。

与以往一样，如果你没有编写自己的赋值操作符，C++会帮你写一个，以便对象相互赋值。默认的C++赋值操作符与其默认的复制行为几乎完全等同，它会递归地从源对象将各个数据成员赋给目标对象。不过，语法稍有一些技巧。

#### 声明一个赋值操作符

以下是另一个SpreadsheetCell类定义，这一次包括了一个赋值操作符：

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(const string& initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
    void setValue(double inValue);
    double getValue();
    void setString(const string& inString);
    string getString();
```

```
protected:
    string doubleToString(double inValue);
    double stringToDouble(const string& inString);

    double mValue;
    string mString;
};
```

赋值操作符类似于复制构造函数，要取源对象的一个 `const` 引用作为参数。在这种情况下，我们称源对象为 `rhs`，这代表等号的“右边”（right-hand side）。调用赋值操作符的对象是等号的左边。

不同于复制构造函数，赋值操作符会返回 `SpreadsheetCell` 对象的一个引用。其原因是，赋值可以串链，如下例所示：

```
myCell = anotherCell = aThirdCell;
```

执行这行代码时，第一件事是调用 `anotherCell` 的赋值操作符，并以 `aThirdCell` 作为“右边”参数。接下来调用 `myCell` 的赋值操作符。不过，参数不是 `anotherCell`。它的右边是前一个赋值（即把 `aThirdCell` 赋给 `anotherCell`）的结果。如果前一个赋值没有返回结果，就不会向 `myCell` 传递任何内容。

你可能不清楚 `myCell` 的赋值操作符为什么不是只取 `anotherCell`。原因是，使用等号只是一个简写，实际上这里完成的是一个方法调用。如果完整地看这行代码的语法，就会发现问题：

```
myCell.operator= (anotherCell.operator= (aThirdCell));
```

现在可以看到，`anotherCell` 的 `operator=` 调用必须返回一个值，这个值要传递给 `myCell` 的 `operator=` 调用。正确的返回值就是 `anotherCell` 本身，因此它可以作为对 `myCell` 赋值的源对象。不过，直接返回 `anotherCell` 效率很低，因此可以返回 `anotherCell` 的一个引用。

实际上可以把赋值操作符声明为返回想要的任何类型，包括 `void`。不过，还是应当返回所调用对象的一个引用，因为这才是客户希望得到的。

### 定义一个赋值操作符

赋值操作符的实现与复制构造函数的实现很相似，但也有一些重要的区别。首先，复制构造函数只在初始化时调用，因此目标对象还没有有效的值。赋值操作符可以重写对象中的当前值。不过，在对象中动态地分配内存之前，这个差别并不会体现出来。有关详细内容请见第 10 章。

其次，C++ 中将对象赋给自己也是合法的。例如，以下代码可以通过编译并运行：

```
SpreadsheetCell cell(4);
cell = cell; // Self-assignment
```

赋值操作符不应禁止自赋值（self-assignment），而且如果出现自赋值的情况，就不应还做完全赋值了。所以，赋值操作符应当在方法开始处检查是否存在自赋值，如果是，则立即返回。

以下是 `SpreadsheetCell` 类中赋值操作符的定义：

```
SpreadsheetCell& SpreadsheetCell::operator=(const SpreadsheetCell& rhs)
{
    if (this == &rhs) {
```

前面这行代码用于检查这是否是自赋值，但有些费解。当等号的左边和右边相同时就会出现自赋值。要分清两个对象是否相同，一种办法是看它们是否占据相同的内存空间，更明确地讲，要看两个对象的

指针是否相等。应该记得, `this` 是对象指针, 该对象的任何方法调用中都可以访问这个指针。因此, `this` 是指向左边对象的指针。类似地, `&rhs` 是指向右边对象的指针。如果两个指针相等, 这个赋值就必然是自赋值。

```
return (*this);  
}
```

`this` 是执行当前方法的对象的指针, 因此 `*this` 就是对象自身。编译器会返回该对象的引用从而与所声明的返回值相匹配。

```
mValue = rhs.mValue;  
mString = rhs.mString;
```

下面是这个方法拷贝的值:

```
return (*this);  
}
```

最后, 如前所述, 它会返回 `*this`。

乍一看覆盖 `operator=` 的语法可能有些奇怪。就像刚开始学习另外一些 C 或 C++ 语法时 (如 `switch` 语句), 你可能也有同样的感觉, 好像这个语法不太对一样。通过 `operator=`, 你已经涉入到一些深层次的语言特性当中了。你实际上改变了 `=` 操作符的含义。遗憾的是, 这种强大的功能需要一些不寻常的语法。不必担心, 你会习惯的!

#### 8.3.4 区别复制和赋值

有时很难区分对象是利用一个复制构造函数初始化得到的, 还是利用赋值操作符赋值得到的。请考虑以下代码:

```
SpreadsheetCell myCell(5);  
SpreadsheetCell anotherCell(myCell);
```

这里 `anotherCell` 是用复制构造函数构造的。

```
SpreadsheetCell aThirdCell = myCell;
```

`aThirdCell` 也是用复制构造函数构造的。这行代码没有调用 `operator=`! 这个语法只是 “`SpreadsheetCell aThirdCell (myCell);`” 的另一种写法。

```
anotherCell = myCell; // Calls operator= for anotherCell.
```

在此 `anotherCell` 已经构造, 因此编译器会调用 `operator=`。

**= 并不一定表示赋值! 当用在变量声明的同一行上时, 它也可能是复制构造的简写。**

#### 对象作为返回值

从函数或方法返回对象时, 有时很难准确地区分究竟发生了复制还是赋值。还记得 `getString()` 的代码吧, 如下所示:

```
string SpreadsheetCell::getString()  
{  
    return (mString);  
}
```



现在来考虑下面的代码：

```
SpreadsheetCell myCell2(5);
string s1;
s1 = myCell2.getString();
```

getString() 返回 mString 时，编译器实际上会调用一个 string 复制构造函数，创建一个匿名的临时 string 对象。将这个结果赋给 s1 时，会以这个临时 string 作为参数，对 s1 调用赋值操作符。然后，这个临时 string 对象会撤销。因此，这样一行代码既调用了复制构造函数，又调用了赋值操作符（分别针对两个不同的对象）。

如果你觉得还不够复杂，可以看看下面这行代码：

```
SpreadsheetCell myCell3(5);
string s2 = myCell3.getString();
```

在这里，getString() 返回 mString 时还是创建了一个临时的匿名 string 对象。不过现在 s2 会调用复制构造函数，而不是赋值操作符。

如果你忘记了这些工作哪个在前哪个在后，或者不清楚调用了哪个构造函数或操作符，可以临时性地在代码中加入一些有帮助的输出，或者利用调试工具逐步跟踪，就可以很容易地得出结论。

#### 复制构造函数和对象成员

还应当注意构造函数中赋值和复制构造函数调用之间的区别。如果一个对象包含其他对象，编译器生成的复制构造函数就会递归地调用所包含的各个对象的复制构造函数。如果你编写了自己的复制构造函数，可以利用初始化列表来提供同样的语义，如前所示。如果在初始化列表中没有包含某个数据成员，编译器会对其完成默认初始化（即调用该对象的 0 参数构造函数），然后才执行构造函数体中的代码。因此，等到执行构造函数体时，所有对象数据成员已经得到了初始化。

也可以不用初始化列表来编写复制构造函数，如下所示：

```
SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell& src)
{
    mValue = src.mValue;
    mString = src.mString;
}
```

不过，在复制构造函数体中为数据成员赋值时，对其使用的是赋值操作符，而不是复制构造函数，因为如前所述，这些数据成员已经得到了初始化。

## 8.4 小结

C++ 为面向对象程序设计提供了一些工具，本章就介绍了这些工具的一些基本方面。在此首先回顾了编写类和使用对象的基本语法，包括访问控制。然后，本章介绍了对象生命期的内容：对象何时构造、撤销和赋值，以及这些动作会调用哪些方法。本章还详细介绍了构造函数的语法，包括初始化列表。在此明确地指出在何种条件下，编译器会编写哪些构造函数，并解释了默认构造函数不带任何参数。

对于某些读者来说，本章的内容大多都是复习。对另外一些人来说，我们希望本章能让你睁开双眼，看看 C++ 面向对象程序设计的世界是什么样子。无论怎样，既然你已经掌握了对象和类，下面可以通过阅读第 9 章来学习更多有关的技巧。

## 第9章 精通类和对象

第8章已经帮助你掌握了类和对象。现在该掌握有关类和对象的一些比较复杂的内容，以便最大限度地使用它们。通过阅读本章，你将学习如何管理和利用 C++ 语言中一些最复杂的方面，编写出安全、有效而且有用的类。

本章提供了一些高级主题的详细教程，包括对象中的动态内存分配，静态 (static) 方法和成员、const 方法和成员、引用和 const 引用成员、方法重载和默认参数、内联 (inline) 方法、嵌套类、友元 (friend)、操作符重载、方法和成员的指针以及接口与实现类的分离。

本章的许多概念都会在高级 C++ 编程，特别是标准模板库中大量出现。

### 9.1 对象中的动态内存分配

有时，在程序实际运行之前，你并不知道将需要多少内存。毫无疑问，对此的解决方案就是在程序执行期间根据需求动态地分配内存。类也不例外。有时在编写类时并不知道一个对象需要多少内存。在这种情况下，对象就应当动态地分配内存。

对象中的动态分配内存会带来一些问题，包括释放内存、处理对象复制以及处理对象赋值。

#### 9.1.1 Spreadsheet 类

第8章介绍了 SpreadsheetCell 类，本章将继续编写电子表格应用中的 Spreadsheet 类。与 SpreadsheetCell 类一样，Spreadsheet 类也将在本章中逐步发展和改进。因此，这里的许多尝试并不一定反映类编写中各个方面的最佳途径。开始时的 Spreadsheet 只是 SpreadsheetCell 的一个二维数组，并提供了方法来设置和获取 Spreadsheet 中特定位置的单元格。尽管大多数电子表格应用都在一个方向上使用字母，而在另一个方向上使用数字来指示单元格，但是这个 Spreadsheet 在两个方向上都使用数字。以下是为一个简单的 Spreadsheet 类所编写的第一个类定义：

```
// Spreadsheet.h
#include "SpreadsheetCell.h"

class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);

    void setCellAt(int x, int y, const SpreadsheetCell& cell);
    SpreadsheetCell getCellAt(int x, int y);

protected:
    bool inRange(int val, int upper);

    int mWidth, mHeight;
    SpreadsheetCell** mCells;
};
```



需要注意的是, Spreadsheet 类中并没有包含 SpreadsheetCell 的一个标准二维数组。相反, 这里包含的是一个 SpreadsheetCell\*\*。其原因在于, 每个 Spreadsheet 对象的两个维可能有所不同, 因此这个类的构造函数必须基于客户指定的高度和宽度动态地分配二维数组。为了动态地分配一个二维数组, 需要编写以下代码:

```
#include "Spreadsheet.h"

Spreadsheet::Spreadsheet(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    mCells = new SpreadsheetCell* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }
}
```

如果栈上有一个名为 s1 的 Spreadsheet, 其宽度为 4、高度为 3, 所得到的内存如图 9-1 所示。

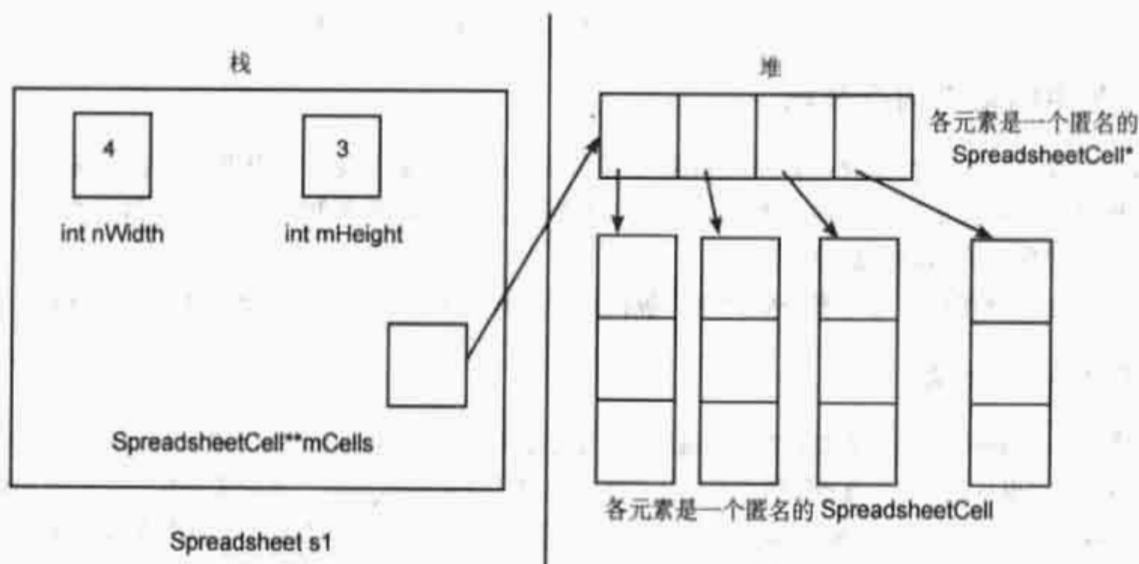


图 9-1

如果不明白这段代码, 请参考第 13 章中介绍内存管理的详细内容。  
设置和获取方法的实现很简单:

```
void Spreadsheet::setCellAt(int x, int y, const SpreadsheetCell& cell)
{
    if (!inRange(x, mWidth) || !inRange(y, mHeight)) {
        return;
    }

    mCells[x][y] = cell;
}

SpreadsheetCell Spreadsheet::getCellAt(int x, int y)
{
    SpreadsheetCell empty;
```

```

    if (!inRange(x, mWidth) || !inRange(y, mHeight)) {
        return (empty);
    }

    return (mCells[x][y]);
}

```

需要注意，这两个方法使用了一个辅助方法 `inRange()` 来检查 `x` 和 `y` 是否分别表示电子表格中合法的坐标。如果试图访问数组中一个非法的字段，就会导致程序出问题。最终的成品应用可能会使用异常来报告错误条件，如第 15 章所述。

### 9.1.2 用析构函数释放内存

如果用完了动态分配的内存，就应该将其释放。如果在对象中动态分配了内存，就应当在析构函数中释放该内存。编译器会确保对象撤销时调用析构函数。以下对前面的 `Spreadsheet` 类定义做了扩展，在此加入了一个析构函数：

```

class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);
    ~Spreadsheet();

    void setCellAt(int x, int y, const SpreadsheetCell& inCell);
    SpreadsheetCell getCellAt(int x, int y);

protected:
    bool inRange(int val, int upper);

    int mWidth, mHeight;
    SpreadsheetCell** mCells;
};

```

析构函数与类同名（也与构造函数同名），但前面有一个波浪线（`~`）。析构函数没有参数，而且一个类只能有一个析构函数。

以下是 `Spreadsheet` 类析构函数的实现：

```

Spreadsheet::~~Spreadsheet()
{
    for (int i = 0; i < mWidth; i++) {
        delete[] mCells[i];
    }

    delete[] mCells;
}

```

这个析构函数会释放在构造函数中分配的内存。不过，并没有要求说析构函数中只能释放内存。你可以在析构函数中写任何代码，不过析构函数仅用于释放内存或释放其他资源是一个不错的想法。

### 9.1.3 处理复制和赋值

在第 8 章中我们了解到，如果没有自己编写复制构造函数和赋值操作符，C++ 就会帮你写一个。这些编译器生成的方法会分别递归地调用对象数据成员上的复制构造函数或赋值操作符。不过，对于基本类型（如 `int`、`double` 和指针），则会提供浅（shallow）或位（bitwise）复制（或赋值）：只是从源对象直

接将数据成员复制或赋值到目标对象。如果在对象中动态分配了内存，就会带来问题。例如，以下代码在将电子表格 s1 传递给 printSpreadsheet() 函数时，会复制 s1 来初始化 s。

```
#include "Spreadsheet.h"

void printSpreadsheet(Spreadsheet s)
{
    // Code omitted for brevity.
}

int main(int argc, char** argv)
{
    Spreadsheet s1(4, 3);
    printSpreadsheet(s1);

    return (0);
}
```

Spreadsheet 包含一个指针变量：mCells。电子表格的浅复制会向目标对象提供 mCells 指针的一个副本，而不是该指针底层数据的副本。这样一来，就会得到以下结果：s 和 s1 都有一个指向相同数据的指针，如图 9-2 所示。

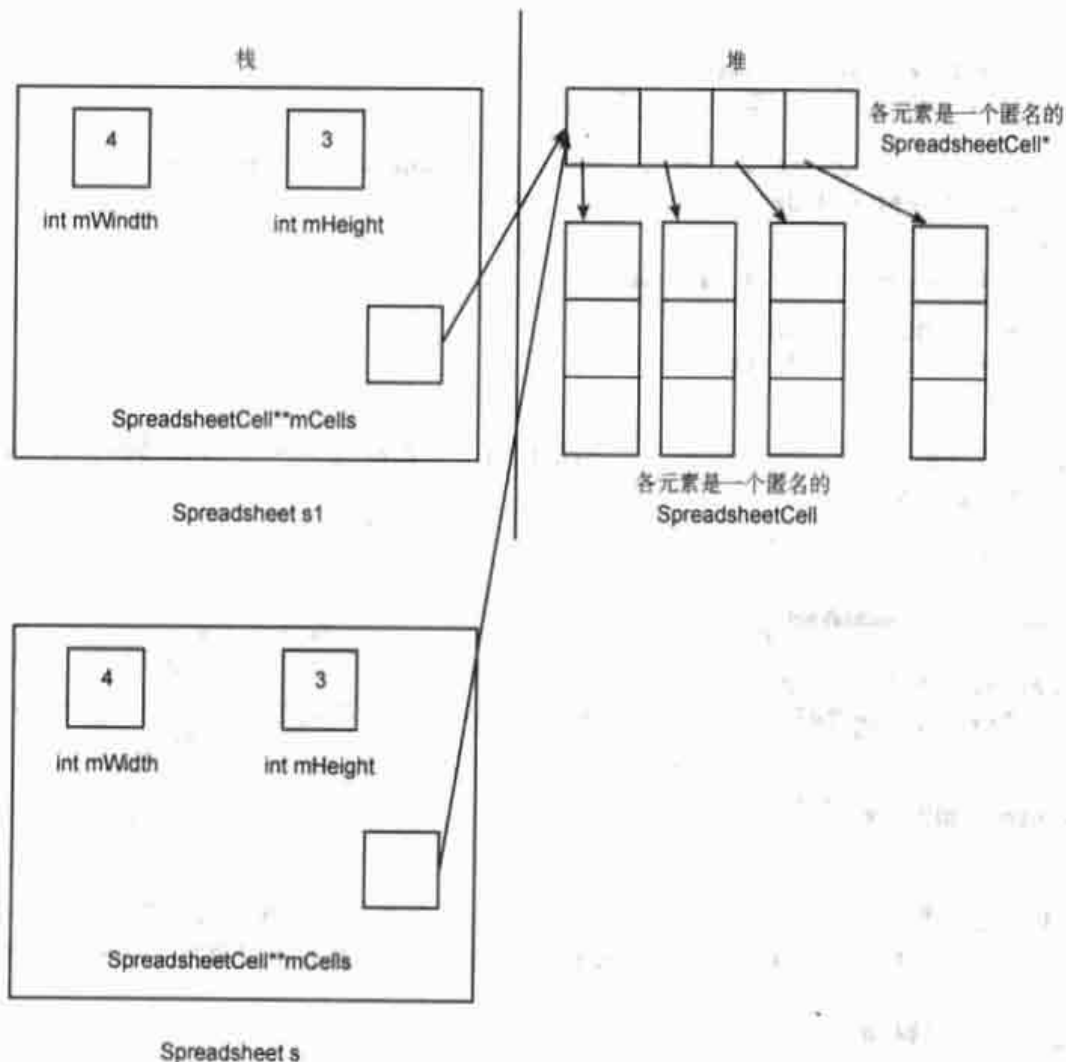


图 9-2

如果 s 要修改 mCells 所指的某个内容，这个修改也会在 s1 中反映出来。更糟糕的是，当 printSpreadsheet() 函数退出时，会调用 s 的析构函数，这会释放 mCells 所指的内存。这样就会带来图 9-3 所示的情况。

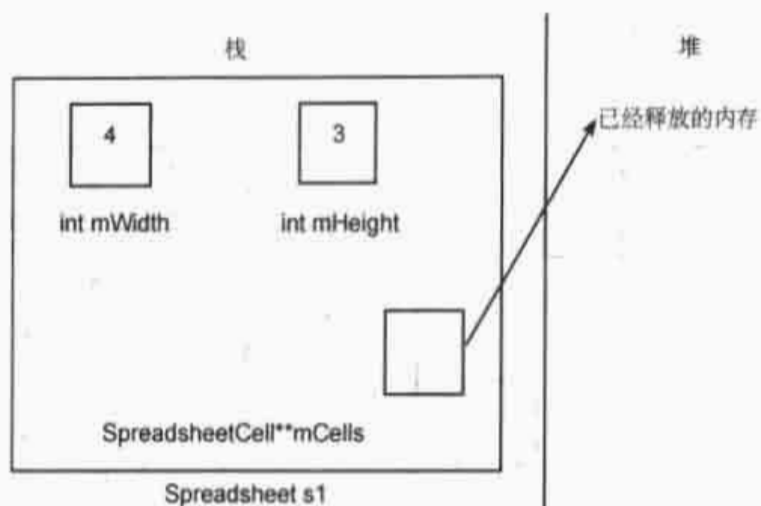


图 9-3

现在 s1 就有一个悬挂指针 (dangling pointer)。

令人难以相信的是，对于赋值，这个问题更严重。假设有以下代码：

```
Spreadsheet s1(2, 2), s2(4, 3);  
s1 = s2;
```

构造了这两个对象后，内存布局如图 9-4 所示。

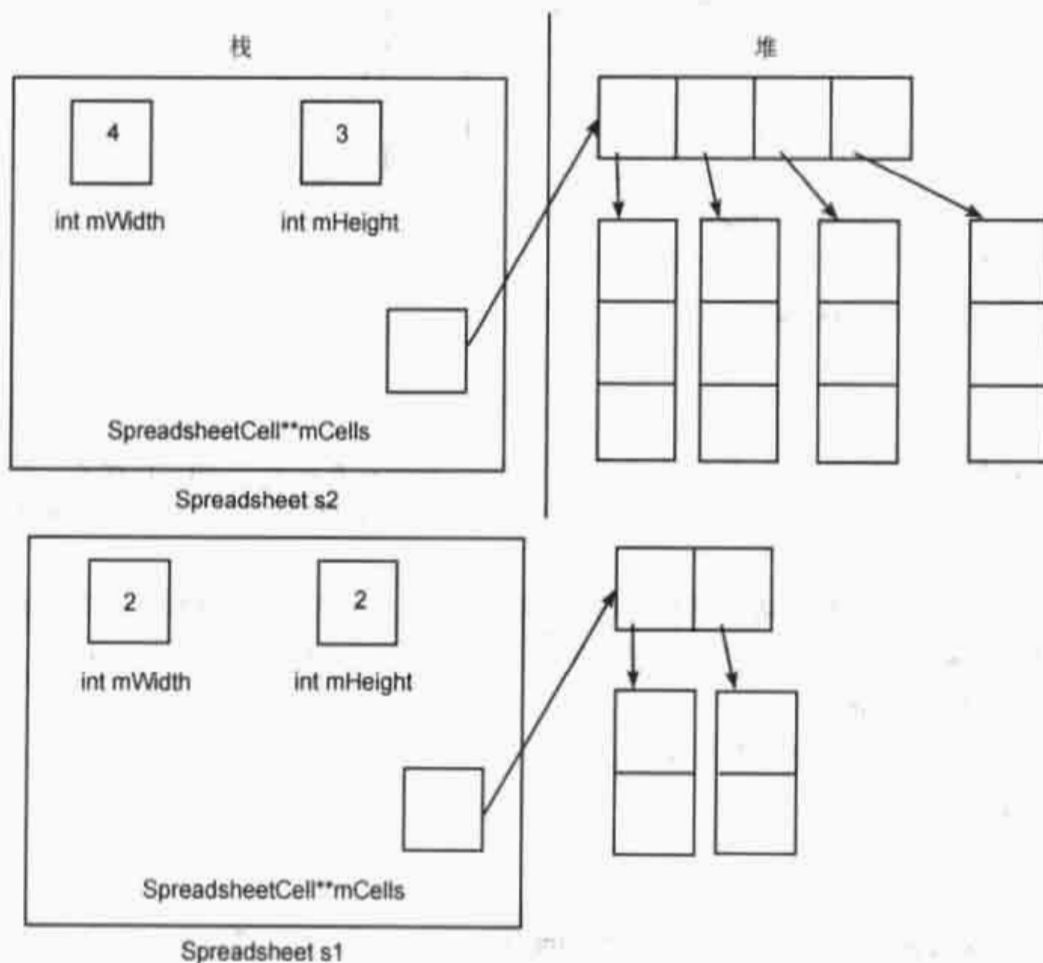


图 9-4

在赋值语句之后，内存布局如图 9-5 所示。

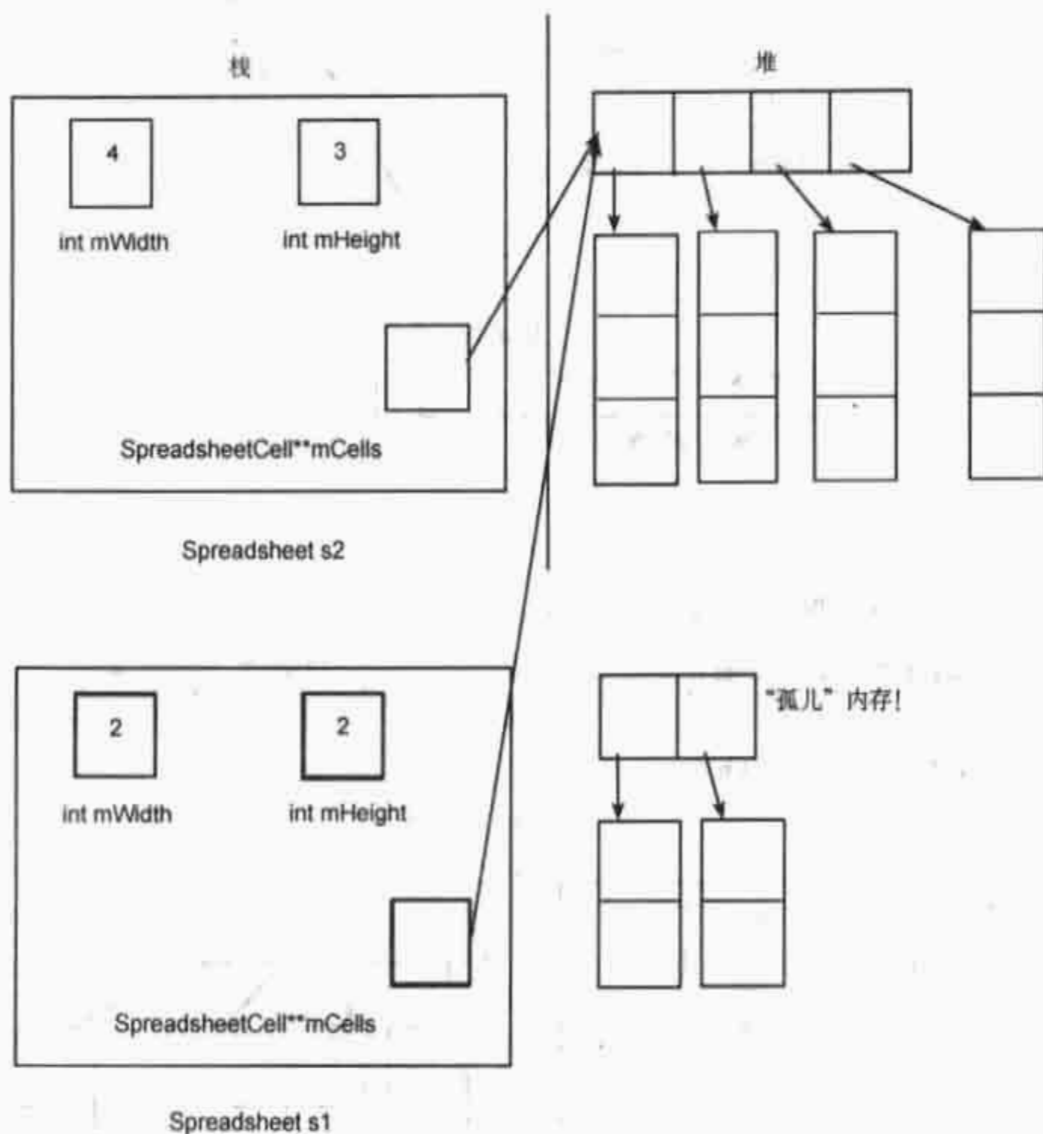


图 9-5

这样一来，不仅 s1 和 s2 中的 mCells 指针都指向相同的内存，而且 s1 中 mCells 原来所指的内存现在成为了“孤儿”（译者注：也称为“野”指针）。这就是为什么在赋值操作符中必须先释放原内存，然后再做深复制的原因。

可以看到，依赖于 C++ 的默认复制构造函数或赋值操作符并不总是一个好想法。只要在类中动态分配了内存，就应该编写自己的复制构造函数来提供内存的深复制（deep copy）。

#### Spreadsheet 复制构造函数

以下是 Spreadsheet 类中对复制构造函数的声明：

```
class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);
    Spreadsheet(const Spreadsheet& src);
```

```

~Spreadsheet();

void setCellAt(int x, int y, const SpreadsheetCell& cell);
SpreadsheetCell getCellAt(int x, int y);

protected:
    bool inRange(int val, int upper);

    int mWidth, mHeight;
    SpreadsheetCell** mCells;
};

```

以下是复制构造函数的定义：

```

Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    int i, j;

    mWidth = src.mWidth;
    mHeight = src.mHeight;

    mCells = new SpreadsheetCell* [mWidth];
    for (i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }

    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}

```

注意，这个复制构造函数复制了所有数据成员，包括 mWidth 和 mHeight，而不只是指针数据成员。复制构造函数中余下的代码为动态分配的二维数组 mCells 提供了一个深复制。

在复制构造函数中复制所有数据成员，而不只是指针成员。

### Spreadsheet 赋值操作符

以下是带赋值操作符的 Spreadsheet 类的定义：

```

class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);
    Spreadsheet(const Spreadsheet& src);
    ~Spreadsheet();

    Spreadsheet& operator=(const Spreadsheet& rhs);

    void setCellAt(int x, int y, const SpreadsheetCell& cell);
    SpreadsheetCell getCellAt(int x, int y);
protected:
    bool inRange(int val, int upper);

    int mWidth, mHeight;
    SpreadsheetCell** mCells;
};

```

以下是 Spreadsheet 类赋值操作符的实现，并分段做了解释。需要注意，要为一个对象赋值时，它应该已经得到了初始化。因此，在分配新内存之前，必须先释放所有动态分配的内存。可以把赋值操作符认为是一个析构函数和一个复制构造函数的组合。在对一个对象赋值时，实际上是使对象“再生”，给它新的生命（数据）。

```
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    int i, j;

    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }
```

以上代码检查是否为自赋值。

```
    // Free the old memory.
    for (i = 0; i < mWidth; i++) {
        delete[] mCells[i];
    }

    delete[] mCells;
```

这段代码相当于析构函数。必须在重新分配内存之前先将原来的所有内存释放掉，否则就会导致内存泄漏。

```
    // Copy the new memory.
    mWidth = rhs.mWidth;
    mHeight = rhs.mHeight;

    mCells = new SpreadsheetCell* [mWidth];
    for (i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }

    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = rhs.mCells[i][j];
        }
    }
}
```

这段代码相当于复制构造函数。

```
    return (*this);
}
```

这个赋值操作符完成了管理对象中动态分配内存的“3大”例程：析构函数、复制构造函数和赋值操作符。如果你发现要编写这3个方法之一，就应该将3个方法全部写出。

只要类会动态分配内存，就需要编写析构函数、复制构造函数和赋值操作符。

### 复制构造函数和赋值操作符的常见辅助例程

复制构造函数和赋值操作符非常相似。因此，将常见任务放在一个辅助方法中通常会很方便。例如，



可以为 Spreadsheet 类增加一个 copyFrom() 方法，然后重新编写复制构造函数和赋值操作符，使它们使用这个辅助方法，如下所示：

```
void Spreadsheet::copyFrom(const Spreadsheet& src)
{
    int i, j;

    mWidth = src.mWidth;
    mHeight = src.mHeight;

    mCells = new SpreadsheetCell* [mWidth];
    for (i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }

    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}

Spreadsheet::Spreadsheet(const Spreadsheet &src)
{
    copyFrom(src);
}

Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    int i;

    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }
    // Free the old memory.
    for (i = 0; i < mWidth; i++) {
        delete[] mCells[i];
    }
    delete[] mCells;

    // Copy the new memory.
    copyFrom(rhs);

    return (*this);
}
```

### 禁止赋值和传值

有时在类中动态分配内存时，最简单的做法就是禁止别人复制或对你的对象赋值。为此，可以将复制构造函数和 operator= 标记为 private。这样一来，如果有人试图按值传递对象，从函数或方法返回对象，或者对其赋值，编译器都会报错。以下 Spreadsheet 类定义禁止了赋值和传值（即按值传递，pass-by-value）：

```

class Spreadsheet
{
    public:
        Spreadsheet(int inWidth, int inHeight);
        ~Spreadsheet();

        void setCellAt(int x, int y, const SpreadsheetCell& cell);
        SpreadsheetCell getCellAt(int x, int y);

    protected:
        bool inRange(int val, int upper);

        int mWidth, mHeight;
        SpreadsheetCell** mCells;

    private:
        Spreadsheet(const Spreadsheet& src);
        Spreadsheet& operator=(const Spreadsheet& rhs);
};

```

编写代码来复制或对 Spreadsheet 对象赋值时，编译器会报出如下的错误消息：“‘=’：cannot access private member declared in class ‘Spreadsheet’”，即“‘=’：不能访问‘Spreadsheet’类中声明的私有成员”。

不必为私有复制构造函数和赋值操作符提供实现。连接器不会寻找其实现，因为编译器不允许代码调用私有复制构造函数和赋值操作符。

## 9.2 不同类型的数据成员

C++ 在数据成员方面提供了许多选择。除了可以在类中声明简单的数据成员之外，还可以创建由类的所有对象所共享的数据成员、引用成员、const 引用成员等。这一节将介绍不同数据成员的复杂的细节内容。

### 9.2.1 静态数据成员

有时，为类的每个对象都提供一个变量副本有些小题大做，甚至不可行。数据成员可能是特定于类的，让每个对象都有自己的副本并不合适。例如，你可能想为每个电子表格提供一个惟一的数字标识符。可能需要一个从 0 计起的计数器，每个新对象都可以从这个计数器获得 ID。这个电子表格计数器实际上属于 Spreadsheet 类，如果让每个 Spreadsheet 对象都有一个副本就不合理了，因为你必须以某种方式让所有计数器保持同步。C++ 利用静态数据成员（static data member）提供了一个解决方案。所谓静态数据成员就是与一个类关联而不是与对象关联的数据成员。可以把静态数据成员认为是特定于一个类的全局变量。以下是一个 Spreadsheet 类的定义，在此包括了一个新的静态计数器数据成员：

```

class Spreadsheet
{
    public:
        // Omitted for brevity
    protected:
        bool inRange(int val, int upper);
        void copyFrom(const Spreadsheet& src);

        int mWidth, mHeight;
        SpreadsheetCell** mCells;
};

```

```
static int sCounter;  
};
```

除了在类定义中列出静态类成员外，还必须在源文件中为其分配空间，这个源文件通常就是类方法定义所在的源文件。可以同时对其初始化，不过请注意，不同于常规的变量和数据成员，静态数据成员默认地会初始化为0。以下是为 sCounter 成员分配空间并完成初始化的代码：

```
int Spreadsheet::sCounter = 0;
```

这个代码出现在所有函数和方法体之外。这就像是在声明一个全局变量，不过 Spreadsheet:: 作用域解析标识指定了这是 Spreadsheet 类的一部分。

### 在类方法中访问静态数据成员

可以把静态数据成员当作常规的数据成员在类方法中使用。例如，你可能希望创建 Spreadsheet 类的一个 mId 成员，并在 Spreadsheet 构造函数中根据 sCounter 成员来进行初始化。以下是加了 mId 成员的 Spreadsheet 类定义：

```
class Spreadsheet  
{  
public:  
    Spreadsheet(int inWidth, int inHeight);  
    Spreadsheet(const Spreadsheet& src);  
    ~Spreadsheet();  
    Spreadsheet& operator=(const Spreadsheet& rhs);  
  
    void setCellAt(int x, int y, const SpreadsheetCell& cell);  
    SpreadsheetCell getCellAt(int x, int y);  
    int getId();  
  
protected:  
    bool inRange(int val, int upper);  
    void copyFrom(const Spreadsheet& src);  
  
    int mWidth, mHeight;  
    int mId;  
    SpreadsheetCell** mCells;  
  
    static int sCounter;  
};
```

下面给出了 Spreadsheet 构造函数的一个实现，其中分配了初始 ID：

```
Spreadsheet::Spreadsheet(int inWidth, int inHeight) :  
    mWidth(inWidth), mHeight(inHeight)  
{  
    mId = sCounter++;  
    mCells = new SpreadsheetCell* [mWidth];  
    for (int i = 0; i < mWidth; i++) {  
        mCells[i] = new SpreadsheetCell[mHeight];  
    }  
}
```

可以看到，构造函数可以访问 sCounter，就好像它是一个常规的成员一样（译者注：即可以把静态数据成员当作常规的实例成员来访问）。要记住，还要在复制构造函数中分配一个 ID：

```

Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    mId = sCounter++;
    copyFrom(src);
}

```

不应在赋值操作符中复制 ID。一旦为一个对象分配了一个 ID，这个 ID 就不能再修改了。

#### 在方法之外访问静态数据成员

访问控制限定符也可以应用于静态 (static) 数据成员：sCounter 是 protected，因此不能从类方法之外访问。

不过，即使它是 protected，还是有一个特殊的情况：可以在源文件中为其声明空间时给它赋一个值，这样的赋值代码不放在任何 Spreadsheet 类方法的内部。再次列出这行声明连带赋值的代码：

```
int Spreadsheet::sCounter = 0;
```

### 9.2.2 const 数据成员

类中的数据成员可以声明为 const，这说明这些成员一经创建和初始化就不能再改动。在对象层次上，常量几乎没有任何意义，因此 const 数据成员通常也是静态的 (static)。当常量仅应用于类时，应当使用 static const 数据成员而不是全局常量。例如，你可能希望为电子表格指定一个最大高度和宽度。如果用户想要用一个更大的高度或宽度（大于你的最大值）构造一个电子表格，则会改为采用最大值。可以把最大高度和宽度置为 Spreadsheet 类的 static const 成员：

```

class Spreadsheet
{
public:
    // Omitted for brevity

    static const int kMaxHeight;
    static const int kMaxWidth;

protected:
    // Omitted for brevity
};

```

由于这些成员是静态的 (static)，必须在源文件中为其声明空间。因为它们是 const，所以这是最后一次为其赋值的机会：

```

const int Spreadsheet::kMaxHeight = 100;
const int Spreadsheet::kMaxWidth = 100;

```

实际上，如果 static const 成员变量是简单类型（如 int 或 char），C++ 标准允许你在类文件中声明这些成员变量的同时对其赋值。

```

class Spreadsheet
{
public:
    // Omitted for brevity

    static const int kMaxHeight = 100;
    static const int kMaxWidth = 100;

protected:
    // Omitted for brevity
};

```

如果以后想在类定义中使用常量，这个功能就很有用。尽管有些较老的编译器不支持这个语法，但是目前大多数都是接受的。实际上，如果在类定义中初始化了静态常量成员，而且未对其完成任何需要实际内存空间的操作（如取其地址），许多编译器都会忽略源文件中静态常量成员的额外定义。

可以在构造函数中使用这些新的常量，如以下代码所示（需要注意三元操作符的使用）：

```
Spreadsheet::Spreadsheet(int inWidth, int inHeight) :
    mWidth(inWidth < kMaxWidth ? inWidth : kMaxWidth),
    mHeight(inHeight < kMaxHeight ? inHeight : kMaxHeight)
{
    mId = sCounter++;
    mCells = new SpreadsheetCell* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new SpreadsheetCell[mHeight];
    }
}
```

kMaxHeight 和 kMaxWidth 是公共的（public），因此可以从程序中的任何位置访问这两个成员，就好像它们是全局变量一样，只是语法稍有不同，必须用作用域解析操作符（::）指定变量是 Spreadsheet 类的一部分：

```
cout << "Maximum height is: " << Spreadsheet::kMaxHeight << endl;
```

### 9.2.3 引用数据成员

Spreadsheet 和 SpreadsheetCell 都很不错，但单凭它们自己并不能成为一个很有用的应用。还需要代码来控制整个电子表格程序，这些代码可以打包到一个 SpreadsheetApplication 类中。

就目前来说，这个类的实现并不重要。现在我们来考虑体系结构问题：电子表格如何与应用通信？应用存储了一个电子表格列表，因此可以与电子表格通信。类似地，每个电子表格应当存储应用对象的一个引用。Spreadsheet 类必须了解 SpreadsheetApplication 类，而不是使用一个完整的 #include，可以只是使用类名的一个超前引用（forward reference），有关详细内容请见第 12 章。以下是一个新的 Spreadsheet 类定义：

```
class SpreadsheetApplication; // forward declaration
```

```
class Spreadsheet
```

```
{
```

```
    public:
```

```
        Spreadsheet(int inWidth, int inHeight,
```

```
                    SpreadsheetApplication& theApp);
```

```
        // Code omitted for brevity.
```

```
    protected:
```

```
        // Code omitted for brevity.
```

```
        SpreadsheetApplication& mTheApp;
```

```
        static int sCounter;
```

```
};
```

需要注意的是，在构造函数中要为各个 Spreadsheet 提供应用引用。如果不指向一个具体的东西，引用是无法存在的，所以必须在构造函数的初始化列表中为 mTheApp 给定一个值。

```

Spreadsheet::Spreadsheet(int inWidth, int inHeight,
    SpreadsheetApplication& theApp)
    : mWidth(inWidth < kMaxWidth ? inWidth : kMaxWidth),
    mHeight(inHeight < kMaxHeight ? inHeight : kMaxHeight), mTheApp(theApp)
{
    // Code omitted for brevity.
}

```

还必须在复制构造函数中初始化引用成员。

```

Spreadsheet::Spreadsheet(const Spreadsheet& src) :
    mTheApp(src.mTheApp)
{
    mId = sCounter++;
    copyFrom(src);
}

```

要记住，一旦初始化了一个引用，就不能再修改它所引用的对象了。因此，不必在赋值操作符中尝试对引用赋值。

#### 9.2.4 const 引用数据成员

常规的引用可以指向 const 对象，类似地，引用成员也可以指向 const 对象。例如，你可能决定 Spreadsheet 只能有应用对象的一个 const 引用。所以可以简单地修改类定义，将 mTheApp 声明为一个 const 引用：

```

class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight,
        const SpreadsheetApplication& theApp);
    // Code omitted for brevity.

protected:
    // Code omitted for brevity.
    const SpreadsheetApplication& mTheApp;

    static int sCounter;
};

```

还有可能存在 static 引用成员，或一个 static const 引用成员，不过一般很少需要用到这种引用成员。

### 9.3 深入了解方法

C++ 也为方法提供了许多选择，这一节将介绍有关的细节内容。

#### 9.3.1 静态方法

与成员一样，方法有时应用于整个类，而不只是单个对象。像编写静态成员一样，也可以编写静态 (static) 方法。举例来说，考虑第 8 章编写的 SpreadsheetCell 类。它有两个辅助方法：stringToDouble() 和 doubleToString()。这些方法并不访问有关特定对象的信息，因此它们可以是静态的。以下是将这些方法置为 static 的类定义：

```

class SpreadsheetCell
{
public:

```



```

        // Omitted for brevity

protected:
    static string doubleToString(double val);
    static double stringToDouble(const string& str);

    // Omitted for brevity
};

```

这两个方法的实现与前面的实现完全相同，甚至不需要在方法定义的前面重复写上 `static` 关键字。不过，要注意静态方法并不是在一个特定对象上调用，因此，它们没有 `this` 指针，而且不能对一个特定对象执行来访问非静态成员。实际上，静态方法就像是一个常规函数。惟一的区别在于，它可以访问类的 `private` 和 `protected` 静态数据成员。

不能在静态方法中访问非静态成员。

在类的任何方法中都可以像调用常规函数一样来调用静态方法。因此，`SpreadsheetCell` 中所有方法的实现都不变。在类之外，需要使用作用域解析操作符为方法名限定一个类名（与静态成员一样）。访问控制也可以正常应用。

你可能想置 `stringToDouble()` 和 `doubleToString()` 为 `public`，这样类之外的其他代码也可以使用这两个方法。倘若如此，可以在任何位置上做如下调用：

```
string str = SpreadsheetCell::doubleToString(5);
```

### 9.3.2 const 方法

`const` 对象就是值不会改变的对象。如果有一个 `const` 对象或对 `const` 对象的引用，编译器就不允许你在该对象上调用任何方法，除非可以保证所调用的方法不会修改任何数据成员。要保证一个方法不会修改数据成员，具体做法就是将方法本身用 `const` 关键字来标记。以下是一个修改后的 `SpreadsheetCell` 类，在此把不修改任何数据成员的方法标记为 `const`：

```

class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(const string& initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
    void setValue(double inValue);
    double getValue() const;
    void setString(const string& inString);
    string getString() const;
    static string doubleToString(double inValue);
    static double stringToDouble(const string& inString);

protected:
    double mValue;
    string mString;
};

```



const 规范（即指定为 const）是方法原型的一部分，在方法定义中也必须带上 const。

```
double SpreadsheetCell::getValue() const
{
    return (mValue);
}
string SpreadsheetCell::getString() const
{
    return (mString);
}
```

将一个方法标记为 const，这表明了与客户代码的一个合约，即保证你不会在此方法中尝试修改对象的内部值。如果把一个方法声明为 const，但它确实会修改数据成员，编译器就会报错。而且不能将静态方法声明为 const，因为这是多余的。静态方法根本没有相应的类实例，因此它们不可能修改内部值。如果方法中对每个数据成员都有一个 const 引用，将此方法置为 const 就很有用。这样一来，如果想修改数据成员，编译器就会提示一个错误。

非 const 对象可以调用 const 和非 const 方法。不过，const 对象只能调用 const 方法。以下是一些例子：

```
SpreadsheetCell myCell(5);

cout << myCell.getValue() << endl; // OK
myCell.setString("6"); // OK

const SpreadsheetCell& anotherCell = myCell;

cout << anotherCell.getValue() << endl; // OK
anotherCell.setString("6"); // Compilation Error!
```

应当养成习惯，将所有不会修改对象的方法都声明为 const，这样就可以在程序中使用 const 对象的引用。

注意，const 对象也可以撤销，而且可以调用其析构函数。不要把析构函数标记为 const。

### 可修改数据成员

有时你会编写一个“逻辑上”是 const 的方法，但实际上它会修改对象的数据成员。这种修改对于用户可见的数据没有任何影响，但是既然从理论上讲是修改，编译器就不允许将此方法声明为 const。例如，假设你想对电子表格应用做一个测量，了解读取数据的频度信息。为此，一种原始的方法是给 SpreadsheetCell 类增加一个计数器，由它对每一个 getValue() 或 getString() 调用计数。遗憾的是，在编译器看来，这就会使这些方法变成非 const 方法，而这并不是你的本意。解决办法就是将新计数器变量置为 mutable，这就会告诉编译器在 const 方法中修改这个变量是允许的。以下是这个新的 SpreadsheetCell 类定义：

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(const string& initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
    void setValue(double inValue);
    double getValue() const;
    void setString(const string& inString);
    string getString() const;
```

```
static string doubleToString(double inValue);
static double stringToDouble(const string& inString);
```

```
protected:
    double mValue;
    string mString;
```

```
mutable int mNumAccesses;
```

```
};
```

下面是 `getValue()` 和 `getString()` 的定义:

```
double SpreadsheetCell::getValue() const
```

```
{
    mNumAccesses++;
    return (mValue);
}
```

```
string SpreadsheetCell::getString() const
```

```
{
    mNumAccesses++;
    return (mString);
}
```

要记住在所有构造函数中都要初始化 `mNumAccesses`!

### 9.3.3 方法重载

你已经注意到, 在一个类中可以写多个构造函数, 这些构造函数都同名。所有这些构造函数的区别仅在于其参数的个数或类型有所不同。对于 C++ 中的任何方法或函数也可以如此。具体地, 可以重载 (overload) 函数或方法名, 即多个函数都同名, 只是这些函数的参数个数或类型有差别。例如, 在 `SpreadsheetCell` 类中, 可以将 `setString()` 和 `setValue()` 都重命名为 `set()`。现在类定义如下所示:

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(const string& initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
    void set(double inValue);
    void set(const string& inString);
    double getValue() const;
    string getString() const;

    // Remainder of the class omitted for brevity
};
```

两个 `set()` 方法的实现仍然保持不变。要注意 `double` 版本的构造函数中原来调用 `setValue()`, 现在必须改为调用 `set()`。编写调用 `set()` 的代码时, 编译器会基于所传递的参数来决定该调用哪一个版本: 如果传递了一个 `string`, 编译器就会调用 `string` 版本; 如果传递的是一个 `double`, 编译器则会调用 `double` 版本。

你可能想对 `getValue()` 和 `getString()` 做同样的工作: 将这两个方法都重命名为 `get()`。不过, 这样将

无法编译。C++ 不允许仅基于方法的返回类型重载一个方法名，因为在许多情况下，编译器无法确定要调用哪一个版本的方法。例如，如果未在任何位置保留方法的返回值，编译器就没有办法区分你到底要调用哪一个版本的方法。

还要注意，可以基于 `const` 重载一个方法。也就是说，可以编写两个同名而且参数相同的方法，只不过其中一个声明为 `const`，而另一个未声明为 `const`。如果你提供了一个 `const` 对象，编译器就会调用 `const` 方法，如果你提供的是一个非 `const` 对象，编译器将调用非 `const` 方法。

### 9.3.4 默认参数

C++ 中提供了一个类似于方法重载的特性，称为默认参数 (default parameter)。可以在原型中为函数和方法参数指定默认值。如果用户指定了实参，就会忽略默认值。如果用户没有指定实参，则使用默认值。不过在此存在一个限制：只能对从最右参数 (rightmost parameter) 开始的一个连续参数表提供默认值。否则，编译器无法将未提供的实参与默认参数匹配。默认参数在构造函数中最有用。例如，可以在 `Spreadsheet` 构造函数中为宽度和高度指定默认值：

```
class Spreadsheet
{
public:
    Spreadsheet(const SpreadsheetApplication& theApp, int inWidth = kMaxWidth,
                int inHeight = kMaxHeight);
    Spreadsheet(const Spreadsheet& src);
    ~Spreadsheet();
    Spreadsheet& operator=(const Spreadsheet& rhs);
    void setCellAt(int x, int y, const SpreadsheetCell& inCell);
    SpreadsheetCell getCellAt(int x, int y);

    int getId();

    static const int kMaxHeight = 100;
    static const int kMaxWidth = 100;

protected:
    // Omitted for brevity
};
```

`Spreadsheet` 构造函数的实现仍保持不变。需要注意，只是在方法声明中指定默认参数，在定义中并不指定。

现在，即使只有一个非复制构造函数，也可以基于一个、两个或三个实参来调用 `Spreadsheet` 构造函数：

```
SpreadsheetApplication theApp;
Spreadsheet s1(theApp);
Spreadsheet s2(theApp, 5);
Spreadsheet s3(theApp, 5, 6);
```

如果一个构造函数的所有参数都有默认值，这个构造函数可以作为默认构造函数。也就是说，可以构建该类的一个对象，而无需指定任何实参。如果想同时声明一个默认构造函数，以及一个多参数构造函数而且其所有参数都有默认值，编译器就会报错，因为如果没有指定任何参数，它就知道该调用哪一个构造函数。

注意，能用默认参数做到的事情也可以利用方法重载办到。你可以写3个不同的构造函数，每个构造函数取的参数个数不同。不过，利用默认参数，只写一个构造函数，就可以完成3种不同的调用，分别提供不同个数的参数。建议你采用自己用得最熟的机制。

### 9.3.5 内联方法

在C++中，你可以建议某个方法或函数调用实际上不应该当作方法或函数调用。实际上，编译器应当把这个方法或函数体直接插入到代码中方法或函数调用所在的位置。这个过程称为内联 (inlining)，需要有此表现的方法或函数称为内联方法或函数。这个过程只是 #define 宏的一个更安全的版本。

在函数或方法定义中，通过将关键字 inline 置于函数或方法名前面，就可以指定一个内联方法或函数。例如，你可能想置 SpreadsheetCell 类的设置和获取方法为内联方法，在这种情况下可以如下定义：

```
inline double SpreadsheetCell::getValue() const
{
    mNumAccesses++;
    return (mValue);
}

inline string SpreadsheetCell::getString() const
{
    mNumAccesses++;
    return (mString);
}
```

现在，编译器就可以选择将 getValue() 和 getString() 调用替换为实际的方法体，而不是生成代码来完成一个函数调用。

在此有一点很重要，所有需要调用内联方法和函数的源文件中，都应该提供内联方法和函数的定义。考虑下面这个问题就能明白为什么要这样了。编译器如果看不到函数定义，它怎么能替换函数体呢？因此，如果你编写了内联函数或方法，就应该在头文件中连同内联函数或方法的原型放上其定义。对于方法，这说明，要把方法定义放在包括类定义的，h 文件中。这样放是很安全的，连接器不会“抱怨”说一个方法有多个定义。从这个意义上讲，这就像是一个 #define 宏。

C++ 还提供了另外一种语法来声明内联方法，其中根本不用 inline 关键字。具体做法是，可以把方法定义直接放在类定义中。以下是采用这种语法的 SpreadsheetCell 类定义：

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(const string& initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
    void set(double inValue);
    void set(const string& inString);

    double getValue() const {mNumAccesses++; return (mValue); }
    string getString() const {mNumAccesses++; return (mString); }

    static string doubleToString(double inValue);
    static double stringToDouble(const string& inString);
}
```

```
protected:
    double mValue;
    string mString;
    mutable int mNumAccesses;
};
```

许多 C++ 程序员都只是知道内联方法语法，并简单地使用，而没有真正理解置方法为内联方法会出现哪些情况。首先，方法能否成为内联方法，这有许多限制。编译器只会内联最简单的方法和函数。如果定义了一个内联方法，但编译器不想内联此方法，则可能只是简单地将此内联指令忽略，但不做任何提示。其次，内联方法可能导致代码膨胀。在所有调用内联方法的地方会到处复制方法体，这就会增加程序可执行文件的大小。因此，应当谨慎使用内联方法和函数。

## 9.4 嵌套类

类定义可以不仅仅包含函数和方法，还可以编写嵌套的类和结构、声明 typedef，或者创建枚举类型。类中声明的所有的一切都在该类的作用域中。如果声明为 public，就可以在类之外对其访问（用 Class-Name:: 作用域解析语法限定其作用域）。

可以在另一个类定义中提供一个类定义。例如，想将 SpreadsheetCell 类定义为 Spreadsheet 类的一部分，则可以如下定义这两个类：

```
class Spreadsheet
{
public:
    class SpreadsheetCell
    {
public:
        SpreadsheetCell();
        SpreadsheetCell(double initialValue);
        SpreadsheetCell(const string& initialValue);
        SpreadsheetCell(const SpreadsheetCell& src);
        SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
        void set(double inValue);
        void set(const string& inString);

        double getValue() const {mNumAccesses++; return (mValue); }
        string getString() const {mNumAccesses++; return (mString); }

        static string doubleToString(double inValue);
        static double stringToDouble(const string& inString);

protected:
        double mValue;
        string mString;
        mutable int mNumAccesses;
    };

    Spreadsheet(const SpreadsheetApplication& theApp, int inWidth = kMaxWidth,
        int inHeight = kMaxHeight);
    Spreadsheet(const Spreadsheet& src);
    ~Spreadsheet();
    Spreadsheet& operator=(const Spreadsheet& rhs);

    // Remainder of Spreadsheet declarations omitted for brevity
};
```

现在, SpreadsheetCell 类在 Spreadsheet 类的内部定义, 因此如果在 Spreadsheet 类之外引用 SpreadsheetCell, 就必须用 Spreadsheet:: 作用域来限定 SpreadsheetCell 类名。这甚至也应用于方法定义。例如, 默认构造函数现在如下所示:

```
Spreadsheet::SpreadsheetCell::SpreadsheetCell() : mValue(0), mNumAccesses(0)
{
}
```

这种语法一下子就变得很笨拙。例如, SpreadsheetCell 赋值操作符的定义如下所示:

```
Spreadsheet::SpreadsheetCell& Spreadsheet::SpreadsheetCell::operator=(
    const SpreadsheetCell& rhs)
{
    if (this == &rhs) {
        return (*this);
    }
    mValue = rhs.mValue;
    mString = rhs.mString;
    mNumAccesses = rhs.mNumAccesses;
    return (*this);
}
```

实际上, 即使对类本身当中的方法, 甚至也要对其返回类型 (而不是参数) 使用这种语法:

```
Spreadsheet::SpreadsheetCell Spreadsheet::getCellAt(int x, int y)
{
    SpreadsheetCell empty;

    if (!inRange(x, mWidth) || !inRange(y, mHeight)) {
        return (empty);
    }
    return (mCells[x][y]);
}
```

可以使用 typedef 将 Spreadsheet:: SpreadsheetCell 重命名为一个更可管理的 SCell, 这样就能避免上述笨拙的语法:

```
typedef Spreadsheet:: SpreadsheetCell SCell;
```

这个 typedef 应当放在 Spreadsheet 类定义之外, 否则必须将 typedef 名本身也用 Spreadsheet:: 限制, 这就会得到 Spreadsheet:: SCell。这么一来就没多大好处了!

现在可以将构造函数写作:

```
SCell::SpreadsheetCell() : mValue(0), mNumAccesses(0)
{
}
```

常规的访问控制也应用于嵌套的类定义。如果声明了一个 private 或 protected 嵌套类, 就只能在外部类内部使用。

应当只对简单的类使用这种嵌套类定义。对于像 SpreadsheetCell 这样的类来说, 嵌套类有些太笨拙, 因此不太适用。



## 9.5 友元

C++ 允许类将其他类或非成员函数声明为友元 (friend)，而且这些友元可以访问 protected 和 private 数据成员和方法。例如，SpreadsheetCell 类可以指定 Spreadsheet 类是它的“友元”，如下所示：

```
class SpreadsheetCell
{
    public:
        friend class Spreadsheet;

        // Remainder of the class omitted for brevity
};
```

现在 Spreadsheet 类的所有方法都可以访问 SpreadsheetCell 类的 private 和 protected 数据成员了。

类似地，可以指定另一个类的一个或多个函数或成员为友元。例如，你可能希望编写一个函数来验证 SpreadsheetCell 对象的数字值和字符串值确实同步。也许希望这个验证例程放在 SpreadsheetCell 类的外面来对一个外部审计进行建模，但是这个函数应当能够访问对象的内部数据成员，以便适当地对其进行检查。以下是带有一个 friend checkSpreadsheetCell() 函数的 SpreadsheetCell 类定义：

```
class SpreadsheetCell
{
    public:
        // Omitted for brevity

        friend bool checkSpreadsheetCell(const SpreadsheetCell &cell);

        // Omitted for brevity
};
```

类中的 friend 声明可以作为函数原型。没有必要在别处写原型（不过这么做也没有坏处）。

以下是此函数定义：

```
bool checkSpreadsheetCell(const SpreadsheetCell &cell)
{
    return (SpreadsheetCell::stringToDouble(cell.mString) == cell.mValue);
}
```

写这个函数与写任何其他函数都是一样的，只不过在此可以直接访问 SpreadsheetCell 类的 private 和 protected 数据成员。不用在函数定义上重复写上 friend 关键字。

friend 类和方法很容易遭到滥用。通过友元，类的内部会暴露给其他类或函数，这就破坏了抽象原则。因此，应当只在有限的情况下（如操作符重载）使用友元。

## 9.6 操作符重载

通常可能希望在对象上完成一些操作，如相加、比较或者作为流向文件写入或读出。例如，只有能对电子表格完成一些算术运算（如完成一整行单元格的累加），电子表格才真正有用。

### 9.6.1 实现加法

采用真正的面向对象方式，SpreadsheetCell 对象应当能够把自身加至其他 SpreadsheetCell 对象。将一个单元格加至另一个单元格，会得到第三个单元格，其值即为相加结果。这不会改变原来的单元格。



完成 SpreadsheetCell 的相加，其含义是对单元格值的相加。在此不考虑字符串表示。

### 第一个尝试：add 方法

可以为 SpreadsheetCell 类声明并定义一个 add 方法，如下所示：

```
class SpreadsheetCell
{
    public:
        // Omitted for brevity

        const SpreadsheetCell add(const SpreadsheetCell& cell) const;

        // Omitted for brevity
};
```

这个方法将两个单元格相加，返回一个新的第三个单元格，其值为前两个单元格值之和。这个方法声明为 const，并取一个 const SpreadsheetCell 的引用作为参数，因为 add() 不会改变原来的两个单元格。它会返回一个 const SpreadsheetCell，因为你不希望用户修改返回值。客户只能将相加得到的结果赋给另一个对象。add() 是一个方法，因此在一个对象上调用，并且要为之传递另一个对象。以下是这个方法的实现：

```
const SpreadsheetCell SpreadsheetCell::add(const SpreadsheetCell& cell) const
{
    SpreadsheetCell newCell;
    newCell.set(mValue + cell.mValue); // call set to update mValue and mString
    return (newCell);
}
```

注意，这个实现创建了一个新的 SpreadsheetCell，名为 newCell，并返回该单元格的一个副本。只有当你为这个类编写了一个复制构造函数时这才奏效。你可能会换种做法，想要返回该单元格的引用。不过，这样做是不可行的，因为一旦 add() 方法结束，newCell 出了作用域，newCell 就会被销毁。你所返回的引用将成为一个悬挂引用。

可以如下使用 add 方法：

```
SpreadsheetCell myCell(4), anotherCell(5);
SpreadsheetCell aThirdCell = myCell.add(anotherCell);
```

这样是可以的，但是有些麻烦。还可以有更好的做法。

### 第二个尝试：重载操作符+作为一个方法

如果能像两个 int 或两个 double 相加那样，用加号来完成两个单元格的相加，就会很方便。如果能做到下面这样就好了。

```
SpreadsheetCell myCell(4), anotherCell(5);
SpreadsheetCell aThirdCell = myCell + anotherCell;
```

幸运的是，C++ 确实允许编写自己的加号，这称为加法操作符，以便用于类。为此，要以 operator+ 命名编写一个方法，如下所示：

```
class SpreadsheetCell
{
    public:
        // Omitted for brevity
```

```
const SpreadsheetCell operator+(const SpreadsheetCell& cell) const;

// Omitted for brevity

};
```

这个方法定义与 add() 方法的实现完全相同：

```
const SpreadsheetCell SpreadsheetCell::operator+(const SpreadsheetCell& cell)
const
{
    SpreadsheetCell newCell;
    newCell.set(mValue + cell.mValue); // Call set to update mValue and mString.
    return (newCell);
}
```

现在就可以像上面那样使用加号来完成两个单元格的相加了！

这个语法很有点让人不适应。不要对这个奇怪的方法名（operator+）有太多顾虑，这个名字与诸如 foo 或 add 之类的方法名没有什么不同。为了理解余下的语法，可以先来看到底发生了什么，这会对理解很有帮助。当 C++ 编译器解析一个程序，并遇到一个操作符时，如 +、-、= 或 <<，它会分别尝试寻找一个名为 operator+、operator-、operator= 或 operator<< 的函数或方法（而且要取适当的参数）。例如，当编译器发现以下这行代码，就会在 SpreadsheetCell 类中寻找一个名为 operator+ 并取另一个 SpreadsheetCell 对象作为参数的方法，或者是一个名为 operator+ 并取两个 SpreadsheetCell 对象作为参数的全局函数：

```
SpreadsheetCell aThirdCell = myCell + anotherCell;
```

注意，并不要求 operator+ 所取的参数必须是同一个类的对象。也可以为 SpreadsheetCell 编写另一个 operator+，并取一个 Spreadsheet 与当前 SpreadsheetCell 相加。这对于程序员来说没有什么意义，但编译器确实允许这样做。

还要注意，可以为 operator+ 提供任何返回值。操作符重载是函数重载的一种形式，应该记得，函数重载不看函数的返回类型。

### 隐式转换

奇怪的是，编写了上述 operator+ 之后，不仅能把两个单元格加在一起，还可以把一个单元格与一个 string、一个 double 或一个 int 相加。

```
SpreadsheetCell myCell(4), aThirdCell;
string str = "hello";

aThirdCell = myCell + str;
aThirdCell = myCell + 5.6;
aThirdCell = myCell + 4;
```

这段代码之所以能正常工作，原因在于，编译器不光会寻找对应指定类型的 operator+，还会做更多工作来找到一个合适的 operator+。编译器还会尝试寻找类型的一个适当转换，以便找到一个适用的 operator+。取相应类型作为参数的构造函数就是合适的转换器。在以上例子中，编译器发现 SpreadsheetCell 试图将其自身加至 double 时，它会寻找一个取 double 作为参数的 SpreadsheetCell 构造函数，并构造一个临时的 SpreadsheetCell 对象传递给 operator+。类似地，当编译器看到代码想把一个 SpreadsheetCell 加到一个 string 时，它会调用 string 版本的 SpreadsheetCell 构造函数来创建一个临时的 Spreadsheet-

Cell 对象传递给 operator+。

这种隐式的转换行为通常很方便。不过，在前面的例子中，将一个 SpreadsheetCell 与一个 string 相加并没有实际的意义。通过使用 explicit 关键字来标记构造函数，可以禁止从一个 string 隐式构造一个 SpreadsheetCell：

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    explicit SpreadsheetCell(const string& initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
    // Remainder omitted for brevity
};
```

explicit 关键字只能放在类定义中，而且只适用于仅有一个参数的构造函数。

### 第三个尝试：全局 Operator+

通过隐式转换，可以使用一个 operator+ 方法将 SpreadsheetCell 对象加至 int 和 double。不过，这个操作符不满足交换律，如以下代码所示：

```
aThirdCell = myCell + 4; // Works fine.
aThirdCell = myCell + 5.6; // Works fine.

aThirdCell = 4 + myCell; // FAILS TO COMPILE!
aThirdCell = 5.6 + myCell; // FAILS TO COMPILE!
```

当 SpreadsheetCell 对象出现在操作符左边时，隐式转换可以很好地工作，但是 SpreadsheetCell 对象位于操作符右边时，就不行了。加法是满足交换律的，因此这里肯定有问题。问题就是 operator+ 方法必须在一个 SpreadsheetCell 对象上调用，而且该对象必须位于 operator+ 的左边。C++ 语言就是这么定义的。如此说来，利用 operator+ 方法是无法让上述代码顺利工作的。

不过，如果把类中的 operator+ 代之以一个全局的 operator+ 函数，这个函数并没有绑定于任何特定对象，这样一来问题就解决了。此函数如下所示：

```
const SpreadsheetCell operator+(const SpreadsheetCell& lhs,
                                const SpreadsheetCell& rhs)
{
    SpreadsheetCell newCell;
    newCell.set(lhs.mValue + rhs.mValue); // Call set to update mValue and mString.
    return (newCell);
}
```

现在上面的 4 行加法代码都能如期正常工作了：

```
aThirdCell = myCell + 4; // Works fine.
aThirdCell = myCell + 5.6; // Works fine.

aThirdCell = 4 + myCell; // Works fine.
aThirdCell = 5.6 + myCell; // Works fine.
```

需要注意，这个全局 operator+ 访问了 SpreadsheetCell 对象的 protected 数据成员。因此，它必须是 SpreadsheetCell 类的一个友元函数：

```
class SpreadsheetCell
{
public:
    // Omitted for brevity

    friend const SpreadsheetCell operator+(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);

    //Omitted for brevity
};
```

你可能想知道如果写了以下这行代码会出现什么情况：

```
aThirdCell = 4.5+5.5;
```

这行代码能编译和运行，但是它没有调用你写的 `operator+`。它只是完成 4.5 和 5.5 的正常相加，然后利用 `double` 版本构造函数构造一个临时的 `SpreadsheetCell` 对象，这个对象会赋给 `aThirdCell`。

事不过三。在 C++ 中，全局 `operator+` 应该就是最佳选择了。

### 9.6.2 重载算术操作符

既然了解了如何编写 `operator+`，其他的基本算术操作符就很好理解了。以下是一、\* 和/ 的声明（还可以重载 %，不过对于 `SpreadsheetCell` 中存储的 `double` 值来说，取模没有意义）：

```
class SpreadsheetCell
{
public:
    // Omitted for brevity

    friend const SpreadsheetCell operator+(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator-(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator*(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator/(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);

    // Omitted for brevity
};
```

以下是重载算术操作符的实现。惟一的难点是：请记住要检查除 0 的情况。如果检测到除 0，这个实现会把结果设置为 0，不过这在数学意义上并不正确：

```
const SpreadsheetCell operator-(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs)
{
    SpreadsheetCell newCell;
    newCell.set(lhs.mValue - rhs.mValue); // Call set to update mValue and mString.
    return (newCell);
}

const SpreadsheetCell operator*(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs)
{
    // ...
```

```

    SpreadsheetCell newCell;
    newCell.set(lhs.mValue * rhs.mValue); // Call set to update mValue and mString.
    return (newCell);
}

const SpreadsheetCell operator/(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs)
{
    SpreadsheetCell newCell;
    if (rhs.mValue == 0) {
        newCell.set(0); // Call set to update mValue and mString.
    } else {
        newCell.set(lhs.mValue / rhs.mValue); // Call set to update mValue
                                                // and mString.
    }
    return (newCell);
}

```

C++不要求乘法非得在 `operator*` 中实现，除法非得在 `operator/` 中实现。完全可以在 `operator/` 中实现乘法，在 `operator+` 中实现除法，如此等等。不过，这么做极易产生混乱，除了想开玩笑，否则这么做是没有道理的。应当尽可能在实现中遵从常用操作符本来的含义。

### 重载算术简写操作符

除了基本的算术操作符外，C++还提供了诸如 `+=` 和 `-=` 等简写操作符。你可能认为，为类写了 `operator+` 也就同时提供了 `operator+=`。可没有这么幸运。你必须显式地重载简写算术操作符。这些操作符与基本算术操作符的区别在于，它们会改变操作符左边的对象，而不是创建一个新对象。还有一个相对次要的差别是，类似于赋值操作符，这些简写算术操作符会生成一个结果，这是修改后对象的一个引用。

简写算术操作符总是要求左边有一个对象，因此应当将其写作方法，而不是全局函数。以下是 `SpreadsheetCell` 类的声明：

```

class SpreadsheetCell
{
public:
    // Omitted for brevity
    friend const SpreadsheetCell operator+(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator-(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator*(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator/(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);

    SpreadsheetCell& operator+=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator-=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator*=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator/=(const SpreadsheetCell& rhs);

    // Omitted for brevity
};

```

实现如下：



```

SpreadsheetCell& SpreadsheetCell::operator+=(const SpreadsheetCell& rhs)
{
    set(mValue + rhs.mValue); // Call set to update mValue and mString.
    return (*this);
}

SpreadsheetCell& SpreadsheetCell::operator-=(const SpreadsheetCell& rhs)
{
    set(mValue - rhs.mValue); // Call set to update mValue and mString.
    return (*this);
}

SpreadsheetCell& SpreadsheetCell::operator*=(const SpreadsheetCell& rhs)
{
    set(mValue * rhs.mValue); // Call set to update mValue and mString.
    return (*this);
}

SpreadsheetCell& SpreadsheetCell::operator/=(const SpreadsheetCell& rhs)
{
    set(mValue / rhs.mValue); // Call set to update mValue and mString.
    return (*this);
}

```

简写算术操作符是基本算术操作符和赋值操作符的组合。有了上述定义，现在就可以写下面的代码了：

```

SpreadsheetCell myCell(4), aThirdCell(2);
aThirdCell -= myCell;
aThirdCell += 5.4;

```

不过，不能写如下的代码（这个限制倒不是坏事！）。

```
5.4 += aThirdCell;
```

### 9.6.3 重载比较操作符

比较操作符（如>、<和==）也是一组可以为类定义的有用的操作符。类似于基本算术操作符，比较操作符应当是全局友元函数，这样对操作符左右两边就都能使用隐式转换了。比较操作符都返回bool值。当然，也可以改变返回类型，不过不建议这么做。以下是 SpreadsheetCell 的声明和定义：

```

class SpreadsheetCell
{
public:
    // Omitted for brevity

    friend const SpreadsheetCell operator+(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator-(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator*(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator/(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
}

```

```

SpreadsheetCell& operator+=(const SpreadsheetCell& rhs);
SpreadsheetCell& operator-=(const SpreadsheetCell& rhs);
SpreadsheetCell& operator*=(const SpreadsheetCell& rhs);
SpreadsheetCell& operator/=(const SpreadsheetCell& rhs);
friend bool operator==(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs);
friend bool operator<(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs);
friend bool operator>(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs);
friend bool operator!=(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs);
friend bool operator<=(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs);
friend bool operator>=(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs);

```

```

// Omitted for brevity
};

```

```

bool operator==(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue == rhs.mValue);
}

bool operator<(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue < rhs.mValue);
}

bool operator>(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue > rhs.mValue);
}

bool operator!=(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue != rhs.mValue);
}

bool operator<=(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue <= rhs.mValue);
}

bool operator>=(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.mValue >= rhs.mValue);
}

```

在有多个数据成员类中，如果对每个数据成员都做比较可能很麻烦。不过，只要实现了==和<，就可以用这两个操作符实现余下的比较操作符。例如，以下是使用operator<实现的operator>=的定义：

```

bool operator>=(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (!(lhs < rhs));
}

```



可以使用这些操作符将 SpreadsheetCell 与其他 SpreadsheetCell 进行比较，还可以与 double 和 int 比较：

```
if (myCell > aThirdCell || myCell < 10) {
    cout << myCell.getValue() << endl;
}
```

#### 9.6.4 利用操作符重载构建类型

很多人发现，操作符重载的语法很复杂，很容易混淆，至少刚开始是这样。而具有讽刺意味的是，采用操作符重载原本是为了让事情更简单。这种简单并非针对编写类的人，而是指使用类的人可以更轻松。关键是让新类尽可能地像一种内置类型（如 int 和 double），比起要记住所应调用的方法名到底是 add() 还是 sum() 来说，使用 + 来完成对象相加更容易一些。

将操作符重载作为一项服务提供给你的类的客户。

到此，你可能不清楚究竟可以重载哪些操作符。答案是：“几乎所有操作符都可以重载，甚至你没听说过的也不例外”。实际上，你对此已经有所接触了。在前一章对象生命期一节中已经看到了赋值操作符，另外已经了解了基本算术操作符、简写算术操作符和比较操作符。重载流插入和析取操作符也很有用。除此以外，有关操作符重载还有一些很复杂但很有趣的事情可以做，你最早也许不会想到还能这么做。STL 大量使用了操作符重载。第 16 章解释了如何以及何时重载余下的操作符。第 21 章到第 23 章将介绍 STL。

### 9.7 方法和成员指针

应该记得，可以创建并使用变量和函数的指针（如果想全面学习指针或函数指针，可以参考第 13 章）。下面，先来考虑类成员和方法的指针。在 C++ 中取类成员和方法的地址来得到指针是完全合法的。不过，要记住，如果没有具体的对象，就不能访问非静态成员或调用非静态方法。类成员和方法的关键就是它们所针对的是各个对象（译者注：这里的类成员和方法是指非静态成员和方法）。因此，如果想通过指针来调用方法或访问成员，必须在对象上下文中对指针解除引用。以下是一个例子。

```
SpreadsheetCell myCell;
double (SpreadsheetCell::*methodPtr) () const = &SpreadsheetCell::getValue;
cout << (myCell.*methodPtr)() << endl;
```

看到这么复杂的语法先不要惊慌。第二行声明了一个名为 methodPtr 的变量，其类型是指向一个 const 方法的指针，这个方法无参数，并返回一个 double。与此同时，还将这个变量初始化为指向 SpreadsheetCell 类的 getValue() 方法。这种语法与声明一个简单的函数指针很相似，只不过在 \*methodPtr 之前增加了 SpreadsheetCell::。这只是说明这个方法指针指向 SpreadsheetCell 类的一个方法。

第三行在 myCell 对象上调用了 getValue() 方法（通过 methodPtr 指针）。请注意这里在 myCell.\*methodPtr 两边的小括号。必须有这对小括号，因为 () 比 \* 的优先级要高。

大多数情况下，C++ 程序员都会使用 typedef 来简化第二行：

```
SpreadsheetCell myCell;
typedef double (SpreadsheetCell::*PtrToGet) () const;
PtrToGet methodPtr = &SpreadsheetCell::getValue;
cout << (myCell.*methodPtr)() << endl;
```

在程序中一般不会出现方法和成员的指针。不过，要记住重要的一点，对于一个指向非静态方法或成员的指针，如果没有对象就不能对其解除引用。你可能想把指向一个非静态方法的指针传递给一个诸如 `qsort()` 之类的函数（它需要一个函数指针），这种想法可不少见，不过这是不会成功的。

注意，即使没有对象，C++ 也允许对一个指向静态成员或方法的指针解除引用。

第 22 章将在 STL 领域进一步讨论方法指针。

## 9.8 构建抽象类

前面已经了解了如何用 C++ 编写类的繁杂语法，再来看第 3 章和第 5 章介绍的设计原则会很有帮助。类是 C++ 中主要的抽象单元。应当对类应用抽象原则，尽可能将其接口与实现分离。具体地，应当置所有数据成员为保护（`protected`）或私有（`private`），并为之提供获取方法和设置方法。`SpreadsheetCell` 类就是这样实现的。`mValue` 和 `mString` 都是保护成员，`set()`、`getValue()` 和 `getString()` 可以获取这些值。这样一来，就可以在内部保持 `mValue` 和 `mString` 同步，而不必担心客户贸然闯入修改这些值。

### 使用接口和实现类

即使采用前面所述的做法和最佳的设计原则，C++ 从根本上来说并未充分保证抽象原则。C++ 语法要求 `public` 接口和 `private`（或 `protected`）数据成员与方法要放在一个类定义中，这样就会将类的一些内部实现细节暴露给它的客户。

将接口和实现分离的好处是，可以让接口更为清晰，又隐藏实现的细节。但不好的地方在于，这可能会带来安全隐患。基本原则是，分别定义两个类：一个接口类，一个实现类。实现类与前面未采用这种方法编写的类是一样的。接口类提供了实现类所提供的公共方法，但它只有一个数据成员，即指向一个实现类对象的指针。接口类方法实现只是调用实现类对象上的相应方法。为了对 `Spreadsheet` 类使用这种方法，只需将原来的 `Spreadsheet` 类重命名为 `SpreadsheetImpl`。以下是新的 `SpreadsheetImpl` 类（它与原来的 `Spreadsheet` 类完全相同，只是名字不同而已）。

```
// SpreadsheetImpl.h
#include "SpreadsheetCell.h"

class SpreadsheetApplication; // Forward reference

class SpreadsheetImpl
{
public:
    SpreadsheetImpl(const SpreadsheetApplication& theApp,
                    int inWidth = kMaxWidth, int inHeight = kMaxHeight);
    SpreadsheetImpl(const SpreadsheetImpl& src);
    ~SpreadsheetImpl();
    SpreadsheetImpl &operator=(const SpreadsheetImpl& rhs);

    void setCellAt(int x, int y, const SpreadsheetCell& inCell);
    SpreadsheetCell getCellAt(int x, int y);

    int getId();

    static const int kMaxHeight = 100;
    static const int kMaxWidth = 100;
```

```
protected:
    bool inRange(int val, int upper);
    void copyFrom(const SpreadsheetImpl& src);

    int mWidth, mHeight;
    int mId;
    SpreadsheetCell** mCells;
    const SpreadsheetApplication& mTheApp;

    static int sCounter;
};
```

接下来定义一个新的 Spreadsheet 类，如下所示：

```
#include "SpreadsheetCell.h"

// Forward declarations
class SpreadsheetImpl;
class SpreadsheetApplication;

class Spreadsheet
{
public:
    Spreadsheet(const SpreadsheetApplication& theApp, int inWidth,
                int inHeight);
    Spreadsheet(const SpreadsheetApplication& theApp);
    Spreadsheet(const Spreadsheet& src);
    ~Spreadsheet();
    Spreadsheet& operator=(const Spreadsheet& rhs);
    void setCellAt(int x, int y, const SpreadsheetCell& inCell);
    SpreadsheetCell getCellAt(int x, int y);
    int getId();

protected:
    SpreadsheetImpl* mImpl;
};
```

这个类现在只包含一个数据成员：指向一个 SpreadsheetImpl 的指针。其 public 方法与原 Spreadsheet 的公共方法相同，只有一个例外：有默认参数的 Spreadsheet 构造函数被分解为两个构造函数，因为原来默认参数的值（译者注：即 kMaxWidth 和 kMaxHeight）是类的成员，但现在 Spreadsheet 类中已经没有这两个成员了。与此不同，SpreadsheetImpl 类会提供默认值。

Spreadsheet 中诸如 setCellAt() 和 getCellAt() 等方法只是将请求继续传给底层 SpreadsheetImpl 对象：

```
void Spreadsheet::setCellAt(int x, int y, const SpreadsheetCell& inCell)
{
    mImpl->setCellAt(x, y, inCell);
}

SpreadsheetCell Spreadsheet::getCellAt(int x, int y)
{
    return (mImpl->getCellAt(x, y));
}

int Spreadsheet::getId()
{
    return (mImpl->getId());
}
```

Spreadsheet 的构造函数必须构造一个新的 SpreadsheetImpl 来完成工作，析构函数则必须释放动态分配的内存。需要注意，SpreadsheetImpl 类只有一个带默认参数的构造函数。Spreadsheet 类的两个构造函数都会调用 SpreadsheetImpl 类的这个构造函数。

```
Spreadsheet::Spreadsheet(const SpreadsheetApplication &theApp, int inWidth,
    int inHeight)
{
    mImpl = new SpreadsheetImpl(theApp, inWidth, inHeight);
}

Spreadsheet::Spreadsheet(const SpreadsheetApplication& theApp)
{
    mImpl = new SpreadsheetImpl(theApp);
}

Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    mImpl = new SpreadsheetImpl(*(src.mImpl));
}

Spreadsheet::~Spreadsheet()
{
    delete (mImpl);
    mImpl = NULL;
}
```

复制构造函数看上去有点奇怪，因为它需要从源电子表格复制底层 SpreadsheetImpl。由于复制构造函数取的是 SpreadsheetImpl 的引用而不是指针，必须将 mImpl 指针解除引用，来得到对象本身，从而使构造函数调用可以得到引用。

Spreadsheet 赋值操作符也必须类似地将赋值传递至底层 SpreadsheetImpl：

```
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    *mImpl = *(rhs.mImpl);
    return (*this);
}
```

赋值操作符的第一行看上去有点奇怪。你可能想这样写：

```
mImpl = rhs.mImpl; // Incorrect assignment!
```

这行代码可以编译，也能运行，但是不会如你所想的那样工作。它只是复制指针，这样左右两边的 Spreadsheet 都拥有指向同一个 SpreadsheetImpl 的指针。如果有一个 Spreadsheet 修改了指针，这个修改就会在另一个对象中反映出来。如果其中一个对象撤销了指针，那么另一个 Spreadsheet 中就会有一个悬挂指针。因此，不能只是赋指针。必须要求 SpreadsheetImpl 赋值操作符运行，但是只有在复制直接的对象时 SpreadsheetImpl 赋值操作符才会起作用。通过对 mImpl 指针解除引用，就能做到直接的对象赋值，这就会调用赋值操作符。需要注意，你只能这样做，因为已经在构造函数中为 mImpl 分配了内存。

这种技术真正地将接口与实现相分离，其功能是相当强大的。尽管刚开始看有些复杂，但是一旦熟悉了，就会发现使用起来很自然。不过，在大多数工作环境中这并不是一个常见的实践做法，因此你可能会发现有的同事会拒绝采纳这种技术。

## 9.9 小结

要编写可靠、设计合理的类，而且要有效地使用对象，本章和第 8 章中已经介绍了为此所需要的所有工具。

你可能已经发现，对象中的动态内存分配会带来一些新的难题，必须在析构函数中释放内存，必须在复制构造函数中复制内存，而在赋值操作符中释放和复制都必须做到。你学习了如何通过一个私有（private）复制构造函数和赋值操作符来避免赋值和传值（按值传递）。

在此还介绍了多种不同类型的数据成员，包括静态（static）成员、const 成员、const 引用成员和可修改成员。你还了解了静态（static）方法、内联（inline）方法和 const 方法，以及方法重载和默认参数。本章还介绍了嵌套类定义和友元（friend）类与函数。

在本章中，你了解了操作符重载，学习了如何重载算术操作符和比较操作符，包括如何重载为全局友元函数和重载为类方法。

最后，你学习了如何提供接口和实现类的分离，从而将抽象发挥到极致。

既然已经熟练掌握了这种面向对象编程语言，下面就可以了解继承和模板了，这些内容将分别在第 10 章和第 11 章中介绍。



## 第 10 章 探索继承技术

没有继承的话，类就只是带有相关行为的数据结构。这本身对于过程性语言来说已经是有力的改善了，不过继承还为编程语言增加了全新的元素。通过继承，可以在现有类的基础上构建新类。这样，类就变成了可重用、可扩展的组件。本章将教给你不同的方法来充分发挥继承的强大功能。你将了解到继承的具体语法以及充分利用继承的高级技术。

读完本章之后，你将了解到：

- 如何通过继承扩展类
- 如何采用继承技术来重用代码
- 如何建立超类和子类之间的交互
- 如何采用继承实现多态性
- 如何使用多重继承
- 如何处理继承中的非常规问题

本章中涉及多态性的内容主要使用了第 8 章和第 9 章讨论的电子表格的例子。如果你还没有阅读这两章，可能需要略读这两章中的示例代码来了解这个例子的背景。本章还引用了第 3 章介绍的面向对象技术，如果你没有阅读第 3 章，而且不熟悉继承的理论，在继续阅读之前，应该先复习一下第 3 章。

### 10.1 使用继承构建类

在第 3 章已经讲到，“is-a”关系表示这样一种模式，即现实世界的对象一般都呈层次体系存在。在程序设计中，如果要在一个类之上构建另一个类，或者只是对一个类做简单的修改来构建另一个类，就与这种模式有关。要实现这个目的，一种途径就是把一个类的代码复制粘贴到另一个类。通过修改相关部分或者添加代码，就可以达到目的，创建一个与原来的类稍有不同的新类。但是由于以下原因，这种方法会让 OOP 程序员愁眉不展，甚至有些懊恼：

- 两个类的代码是完全分离的，所以即使在原来的类中修正了 bug，新类也不会反映出这种 bug 修正。
- 编译器并不知道两个类之间的相互关系，所以这两个类不是多态的——它们并非同一个事物的不同变种。
- 这种方法没有建立真正的 is-a 关系。新类与原来的类相似，这只是因为新类共享了原类的代码，并不是因为新类与原类确实是同一种对象类型。
- 有时候并不能得到原始代码。原始代码或许只是以预编译的二进制格式存在，所以复制粘贴代码不大现实。

毫无疑问，C++ 对定义真正的 is-a 关系提供了内置支持。本章后面的部分将介绍 C++ is-a 关系的特性。

#### 10.1.1 扩展类

在 C++ 中编写类定义时，可以告诉编译器，要定义的要继承自（inheriting from）或扩展（extending）一个现有的类。这样，自己定义的类将会自动包含原始类的数据成员和方法，原始类称为父类（parent class）或超类（superclass）。扩展现有类的类称为派生类（derived class）或者子类（subclass），



通过扩展现有的类，这个类就能够只描述与父类不同的方面。

在 C++ 中扩展一个类时，在编写类定义时要指定所扩展的类。为了说明继承的语法，在此定义了两个类，分别名为 Super 和 Sub。不必担心——后面还有更多有趣的例子呢！首先来看一下类 Super 的定义。

```
class Super
{
    public:
        Super();

        void someMethod();

    protected:
        int mProtectedInt;

    private:
        int mPrivateInt;
};
```

要想定义一个继承类 Super 的新类 Sub，就要告诉编译器 Sub 是从 Super 派生来的，语法如下：

```
class Sub : public Super
{
    public:
        Sub();

        void someOtherMethod();
};
```

Sub 本身是一个完整的类，只不过共享了类 Super 的特性。现在先不要理会上面代码中的 public 一词——随后会在本章中解释它的含义。图 10-1 给出了类 Sub 与 Super 之间的简单关系。现在可以像声明其他对象一样声明 Sub 类型的对象了。甚至可以定义第三个类作为 Sub 的子类，从而形成类的一个链，如图 10-2 所示。

Sub 不见得是 Super 惟一的子类，Super 也可以有其他子类，这些子类则成为 Sub 的兄弟（sibling）类，如图 10-3 所示。

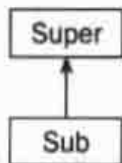


图 10-1

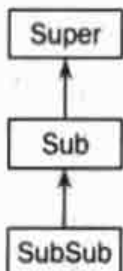


图 10-2

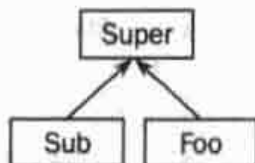


图 10-3

### 继承的客户视图

对于客户代码或者你的代码的其他部分来说，Sub 类型的对象也是 Super 类型的对象，因为 Sub 是从 Super 继承来的。也就是说，对于这些代码来说，类 Super 和 Sub 的所有 public 方法和数据成员都是可用的。

如果使用子类的代码要调用一个方法，它无需知道是继承链中的哪一个类定义了该方法。比如下面这段代码就调用了 Sub 对象的两个方法，虽然其中一个方法是由 Super 类定义的。

```
Sub mySub;  
  
mySub.someMethod();  
mySub.someOtherMethod();
```

了解继承的单向性很重要。类 Sub 对于类 Super 有非常明确的关系，但是类 Super 对于类 Sub 一无所知。也就是说 Super 类型的对象不支持 Sub 的 public 方法和数据成员，因为 Super 不是 Sub。

由于类 Super 没有定义 public 方法 someOtherMethod()，因此下面这段代码不能编译。

```
Super mySuper;  
  
mySuper.someOtherMethod(); // BUG! Super doesn't have a someOtherMethod().
```

从其他代码的角度来看，属于子类的对象也属于其超类。

指向对象的指针或者引用可以引用所声明类的对象或其任何子类的对象。这个比较棘手的话题随后会在本章中详细解释。在此要理解的概念是，指向 Super 的指针可以实际上指向 Sub 对象。对于引用也是这样。客户代码仍然只访问 Super 中的方法和数据成员，但是通过这种机制，对 Super 进行操作的代码也可以对 Sub 进行操作。

例如，虽然下面这段代码开始会出现类型不匹配问题，但是可以编译成功，也能良好运行。

```
Super* superPointer = new Sub(); // Create a sub, and store it in a super pointer.
```

### 继承的子类视图

对于子类本身，如何编写或者有何行为，这与一般的类并没有什么变化。像常规的类一样，在子类上也可以定义方法和数据成员。前面定义的类 Sub 声明了一个方法 someOtherMethod()。因而，类 Sub 通过增加一个额外的方法扩展了类 Super。

子类可以访问其超类的 public 和 protected 方法以及数据成员，就像是子类自己的方法和数据成员一样，因为从理论上讲，确实如此。例如，在类 Sub 中方法 someOtherMethod() 的实现可以使用在类 Super 中声明的数据成员 mProtectedInt。以下代码显示了这个实现。对超类数据成员或方法的访问都是一样的，就好像这个数据成员或方法声明为子类中的一部分一样。

```
void Sub::someOtherMethod()  
{  
    cout << "I can access the superclass data member mProtectedInt." << endl;  
    cout << "Its value is " << mProtectedInt << endl;  
}
```

第 8 章介绍访问限定符 (public、private 和 protected) 时，很有可能会混淆 private 与 protected 之间的区别。现在了解了子类的概念，两者之间的区别就应该清楚了。如果类把方法或者数据成员声明为 protected，子类可以访问这些方法和数据成员。如果声明为 private，子类则不能访问。

以下方法 someOtherMethod() 的实现代码中，子类试图访问超类的 private 数据成员，所以不能编译成功。

```
void Sub::someOtherMethod()  
{  
    cout << "I can access the superclass data member mProtectedInt." << endl;  
    cout << "Its value is " << mProtectedInt << endl;  
  
    cout << "The value of mPrivateInt is " << mPrivateInt << endl; // BUG!  
}
```

访问限定符 `private` 能对类和潜在的子类之间如何交互有所控制。在实际中，大部分的数据成员都声明为 `protected`，而方法则大部分声明为 `public` 或者 `protected`。因为在多数情况下，你或者你的同事可能要扩展这些类，把方法或者数据成员声明为 `private` 会把这些潜在的应用拒之门外，其实你并不想这样。当然，偶尔也会使用 `private` 来防止子类访问可能危险的方法。在编写可能会被外部或者未知的第三方扩展的类时，`private` 限定符也是有用的，这样可以阻挡一些访问以防止误用。

从子类的角度来看，超类中所有的 `public` 和 `protected` 数据成员和方法都是可用的。

### 10.1.2 覆盖方法

第 3 章已经介绍过，类继承的主要原因是增加或替换一些功能。类 `Sub` 增加了一个方法 `someOtherMethod()`，给父类增加了功能。而另一个方法 `someMethod()` 是从类 `Super` 继承的，它在子类中的行为与在超类中的行为是完全相同的。很多情况下需要替换或者覆盖（overriding）方法来修改类的行为。

**不要犹豫，将所有方法都声明为 `virtual`**

在 C++ 中覆盖方法走了一个小小的弯路，即必须使用关键字 `virtual`。超类中只有声明为 `virtual` 的方法才能被子类正确覆盖。关键字 `virtual` 要放在方法声明的开头，如下所示，这是修改后的类 `Super`。

```
class Super
{
    public:
        Super();

        virtual void someMethod();

    protected:
        int mProtectedInt;

    private:
        int mPrivateInt;
};
```

关键字 `virtual` 有些微妙，它常被视为 C++ 中一个拙劣的设计。一个好的经验就是把所有的方法都声明为 `virtual`。这样就不用担心覆盖方法是否有效了。惟一的不足就是性能会受到一些影响。关键字 `virtual` 的微妙之处将在本章的结尾介绍，其性能在第 17 章再进一步讨论。

虽然将来未必会对类 `Sub` 进行扩展，但是将方法也声明为 `virtual` 是一个不错的想法，以备将来真的要扩展。

```
class Sub : public Super
{
    public:
        Sub();

        virtual void someOtherMethod();
};
```

作为一条经验，要把所有的方法都用 `virtual` 声明（包括析构函数，但是不包括构造函数），来避免因遗漏关键字 `virtual` 而产生的相关问题。

#### 覆盖方法的语法

要覆盖一个方法，可以像在超类中声明一样，简单地在子类的类定义中重新声明。在子类的实现文

件中，要为该方法提供一个新定义。

例如，类 Super 有一个方法 someMethod()。someMethod() 的定义在文件 Super.cpp 中给出，如下所示：

```
void Super::someMethod()
{
    cout << "This is Super's version of someMethod()." << endl;
}
```

注意，在方法的定义前面不必重复关键字 virtual。

如果希望在类 Sub 中提供方法 someMethod() 的新定义，首先必须在 Sub 的类定义中增加该方法，如下所示：

```
class Sub : public Super
{
    public:
        Sub();

        virtual void someMethod(); // Overrides Super's someMethod()
        virtual void someOtherMethod();
};
```

方法 someMethod() 的新定义与类 Sub 的其他方法一起声明。

```
void Sub::someMethod()
{
    cout << "This is Sub's version of someMethod()." << endl;
}
```

### 覆盖方法的客户视图

经过前面的修改之后，其他的代码仍然像以前一样调用 someMethod()。而且该方法可以在类 Super 的对象上调用，也可以在类 Sub 的对象上调用。但是在不同类的对象上，someMethod() 的行为会有所不同。

例如，下面的代码调用了 Super 的 someMethod()，它的功能和前面一样。

```
Super mySuper;

mySuper.someMethod(); // Calls Super's version of someMethod().
```

这段代码的输出结果是：

```
This is Super's version of someMethod().
```

如果这段代码声明了一个类 Sub 的对象，那么会自动调用 someMethod() 方法的另一个实现。

```
Sub mySub;

mySub.someMethod(); // Calls Sub's version of someMethod()
```

这一次的输出结果则是：

```
This is Sub's version of someMethod().
```

类 Sub 的对象的其他方面则保持不变。从 Super 继承的其他方法仍然使用 Super 的定义，在 Sub 中明

确覆盖的方法除外。

像前面学到的一样，指针或引用可以引用类的对象或任意子类的对象。对象自身“很清楚”它实际上是哪个类的成员，所以只要方法用 virtual 声明，就会调用正确的方法。例如，如果一个 Super 引用实际上指向一个 Sub 对象，那么调用 someMethod() 方法实际上是调用 Sub 的 someMethod() 方法，如下所示。若在超类中遗漏了关键字 virtual，则不能进行正确地覆盖。

```
Sub mySub;
Super& ref = mySub;

ref.someMethod(); // Calls Sub's version of someMethod()
```

记住一点，即使超类的引用或者指针知道实际上它是一个子类，也不能调用在超类中未定义的子类方法或者成员。下列代码不能成功编译，因为 Super 引用并没有包含方法 someOtherMethod()。

```
Sub mySub;
Super& ref = mySub;

mySub.someOtherMethod(); // This is fine.
ref.someOtherMethod();   // BUG
```

对于非指针、非引用的对象，则不具有这个特性，它不知道自己到底是哪个子类。因为 Sub 是 Super，所以可以对 Sub 进行类型强制转换或者把 Sub 赋值给 Super。然而，这样一来，对象就会丢失子类的一些知识。

```
Sub mySub;
Super assignedObject = mySub; // Assign Sub to a Super.

assignedObject.someMethod(); // Calls Super's version of someMethod()
```

要想记住这个看起来有点奇怪的行为，一种方法就是想像对象在内存中的样子。把 Super 对象画成一个盒子，这个盒子占据一定的内存。Sub 对象是一个稍微大点的盒子，因为它具备 Super 的所有内容，并且增加了自己的一些内容。在使用 Sub 的引用或者指针时，盒子不会改变——只是采用了一种新的办法来访问而已。然而，在把 Sub 强制转换为 Super 时，就得扔掉一些类 Sub 独有的内容，才能把它放进一个较小的盒子里。

超类指针或引用在引用子类时，子类仍然会保留它们覆盖的方法。而在强制转换为超类对象时，子类会丧失了它们的独有特性。覆盖方法和子类数据的丢失称为切割 (slicing)。

## 10.2 继承以实现重用

现在你已经熟悉了继承的基本语法，下面就来分析继承为什么是 C++ 的重要特征。就像在第 3 章读到的那样，继承是一种载体，它可以让你充分利用既有的代码。本节就给出了继承在代码重用中的一个实际应用。

### 10.2.1 类 WeatherPrediction

假设给你分配了一个任务，要编写一段程序来发布简单的天气预报。天气预报的相关知识可能超越了程序员的专业知识范围，所以在此可以获得一个第三方类库，该类库编写时的理论基础是，天气预报是根据当前气温和木星与火星之间的当前距离预报的（哈，看起来还是蛮真实的）。为了保护预测算法的



知识产权, 这个第三方包是作为已编译的库发布的, 但是你可以得到类的定义。类 WeatherPrediction 的定义如下所示。

```
// WeatherPrediction.h

/**
 * Predicts the weather using proven new-age
 * techniques given the current temperature
 * and the distance from Jupiter to Mars. If
 * these values are not provided, a guess is
 * still given but it's only 99% accurate.
 */
class WeatherPrediction
{
public:
    virtual void setCurrentTempFahrenheit(int inTemp);
    virtual void setPositionOfJupiter(int inDistanceFromMars);

    /**
     * Gets the prediction for tomorrow's temperature
     */
    virtual int getTomorrowTempFahrenheit();

    /**
     * Gets the probability of rain tomorrow. 1 means
     * definite rain. 0 means no chance of rain.
     */
    virtual double getChanceOfRain();

    /**
     * Displays the result to the user in this format:
     * Result: x.xx chance. Temp. xx
     */
    virtual void showResult();

protected:
    int mCurrentTempFahrenheit;
    int mDistanceFromMars;
};
```

该类解决了这个天气预报程序的大部分问题。但是, 像往常一样, 现有的类库并不能正好满足需求。首先, 这里给出的气温都是华氏温度。而程序需要使用摄氏温度。而且 showResult() 方法并没有生成十分友好的用户界面。给用户一些比较友好的信息应该会好些。

### 10.2.2 在子类中增加功能

在第 3 章学习继承时, 首先介绍的技术是增加功能。从根本上说, 程序需要某些类似类 WeatherPrediction 的内容, 但是另外还需要几个额外的小功能。这看上去是一个通过继承来重用代码的好例子。我们先来定义一个从类 WeatherPrediction 继承的新类 MyWeatherPrediction。

```
// MyWeatherPrediction.h

class MyWeatherPrediction : public WeatherPrediction
{
};
```



上面的类定义可以很好地编译通过。类 `MyWeatherPrediction` 可以代替类 `WeatherPrediction`。它能提供相同的功能，但是尚未增加新的功能。

作为第一个修改，可能需要给类 `MyWeatherPrediction` 增加摄氏温度的相关内容。在此你可能会感到一点迷惑，因为你并不知道这个类的内部在做什么。如果内部计算使用的都是华氏温度，那么如何增加对摄氏度的支持呢？一种方法就是使用这个子类作为桥梁，在可能使用这两种度量方法的用户和只支持华氏温度的超类之间建立一个接口。

为了支持摄氏温度，第一步是给该类增加一些新方法，允许客户以摄氏度而不是华氏度来设置温度，并得到以摄氏度表示的明天的天气预报。同时还需要一些 `protected` 辅助方法进行摄氏度和华氏度之间的转换。这些方法可以是静态的，因为它们对该类的所有实例都相同。

```
// MyWeatherPrediction.h
```

```
class MyWeatherPrediction : public WeatherPrediction
{
public:
    virtual void setCurrentTempCelsius(int inTemp);

    virtual int getTomorrowTempCelsius();

protected:
    static int convertCelsiusToFahrenheit(int inCelsius);
    static int convertFahrenheitToCelsius(int inFahrenheit);
};
```

新方法的命名约定和父类相同。记住一点，从其他代码的角度来看，`MyWeatherPrediction` 对象包含了类 `MyWeatherPrediction` 和 `WeatherPrediction` 中定义的所有功能。采用与父类相同的命名约定，就可以提供一个一致的接口。

这里把摄氏/华氏温度转换方法留给读者作为练习——这是一件很好玩的工作！其他的两个方法更有意思。要用摄氏度设置当前温度，首先要将温度进行转换，然后以父类理解的单位（即度量方式）提交给父类。

```
void MyWeatherPrediction::setCurrentTempCelsius(int inTemp)
{
    int fahrenheitTemp = convertCelsiusToFahrenheit(inTemp);
    setCurrentTempFahrenheit(fahrenheitTemp);
}
```

可以看到，一旦对温度进行了转换，该方法就会简单地从超类调用已有的功能。同样，方法 `getTomorrowTempCelsius()` 的实现也使用了父类已有的功能来得到华氏温度，但是在返回之前会对结果进行转换。

```
int MyWeatherPrediction::getTomorrowTempCelsius()
{
    int fahrenheitTemp = getTomorrowTempFahrenheit();
    return convertFahrenheitToCelsius(fahrenheitTemp);
}
```

这两个新方法只是简单“包装”了现有的功能，提供了一个新的接口来使用父类，这样就有效地重用了父类。

当然，你也可以增加与父类现有功能完全不相干的新功能。比如，可以增加一个方法从 Internet 上

获取两种方式的天气预报, 或者根据预报的天气建议行程。

### 10.2.3 在子类中进行功能替换

建立子类的另一种主要技术是替换现有的功能。类 WeatherPrediction 的方法 showResult() 就非常需要修改。类 MyWeatherPrediction 可以覆盖该方法来使用自己的实现取代父类的这个行为。

类 MyWeatherPrediction 的新定义如下。

```
// MyWeatherPrediction.h

class MyWeatherPrediction : public WeatherPrediction
{
public:
    virtual void setCurrentTempCelsius(int inTemp);

    virtual int getTomorrowTempCelsius();

    virtual void showResult();

protected:
    static int convertCelsiusToFahrenheit(int inCelsius);
    static int convertFahrenheitToCelsius(int inFahrenheit);
};
```

一个用户友好的新实现如下。

```
void MyWeatherPrediction::showResult()
{
    cout << "Tomorrow's temperature will be " <<
        getTomorrowTempCelsius() << " degrees Celsius (" <<
        getTomorrowTempFahrenheit() << " degrees Fahrenheit)" << endl;

    cout << "The chance of rain is " << (getChanceOfRain() * 100) << " percent"
        << endl;

    if (getChanceOfRain() > 0.5) {
        cout << "Bring an umbrella!" << endl;
    }
}
```

对于使用该类的客户程序来说, 就好似原来的 showResult() 方法从未存在一样。只要对象是一个 MyWeatherPrediction 对象, 调用的就是方法的新版本 (即子类中的方法)。

这样一修改, 类 MyWeatherPrediction 就成为了包含新方法的一个新类, 它有更具体的目的。但是它充分利用了超类现有的功能, 所以并不需要太多的代码。

## 10.3 考虑父类

编写子类时, 需要了解父类和子类之间的交互。比如创建顺序、构造函数链和强制类型转换等问题都可能成为潜在的 bug 源头。

### 10.3.1 父构造函数

对象不会一下子突然出现, 它们必须同父类以及它们包含的对象一起构造。C++ 定义的对象创建顺序如下:

1. 如果有的话，首先构造基类。
2. 非 static 数据成员按照声明的顺序构造。
3. 执行构造函数体。

这些规则可以递归应用。如果类有祖父类，则要在父类之前进行初始化，依此类推。下面的代码展示了这种创建顺序。需要提醒的是，我们一般反对内联方法，后面的代码就没有使用内联方法。但是为了例子的可读性和简洁性，在这个例子中我们没有遵循这条原则。下面代码执行正确的话将输出结果 123。

```
#include <iostream>
using namespace std;

class Something
{
public:
    Something() { cout << "2"; }
};

class Parent
{
public:
    Parent() { cout << "1"; }
};

class Child : public Parent
{
public:
    Child() { cout << "3"; }

protected:
    Something mDataMember;
};

int main(int argc, char** argv)
{
    Child myChild;
}
```

创建 myChild 对象时，首先调用 Parent 的构造函数，结果输出字符串“1”。接下来调用 Something 构造函数初始化 mDataMember，该函数输出字符串“2”。最后调用 Child 构造函数，输出 3。

注意，Parent 构造函数是自动调用的。如果父类存在默认的构造函数，C++ 会自动调用。若父类没有默认的构造函数，或者尽管有，但是希望使用另外一个，则可以把构造函数用链串起来，就像初始函数列表中初始化数据成员一样。

下面的代码给出了没有默认构造函数的 Super 类。相关版本的 Sub 必须明确告知编译器如何调用 Super 构造函数，否则代码不能通过编译。

```
// Super.h
class Super
{
public:
    Super(int i);
};
```

```
// Sub.h
class Sub : public Super
{
    public:
        Sub();
};

// Sub.cpp
Sub::Sub() : Super(7)
{
    // Do Sub's other initialization here.
}
```

在前面这段代码中，Sub 构造函数传递一个常量（7）给 Super 构造函数。如果构造函数需要参数，Sub 也可以传递一个变量：

```
Sub::Sub (int i) : Super (i) {}
```

从子类向超类传递构造函数参数是很好的做法，也很正常。但是传递数据成员则是不行的。尽管代码可以通过编译，但是要在构造超类之后才能初始化数据成员。如果把数据成员作为参数传递给父构造函数，则是未初始化的。

### 10.3.2 析构函数

因为析构函数不能包含参数，所以 C++ 会给父类自动调用析构函数。撤销的顺序恰好与构造顺序相反。

1. 调用析构函数体。
2. 按照构造的逆序删除数据成员。
3. 如果有父类，删除父类。

这些规则也可以递归应用。链中的最后一个成员总是首先被删除。下面的代码为前面的例子增加了析构函数。正确执行的话将输出“123321”。

```
#include <iostream>
using namespace std;

class Something
{
    public:
        Something() { cout << "2"; }
        virtual ~Something() { cout << "2"; }
};

class Parent
{
    public:
        Parent() { cout << "1"; }
        virtual ~Parent() { cout << "1"; }
};

class Child : public Parent
{
    public:
        Child() { cout << "3"; }
```

```
virtual ~Child() { cout << "3"; }
```

```
protected:
    Something mDataMember;
};
```

```
int main(int argc, char** argv)
{
    Child myChild;
}
```

注意，析构函数全部使用关键字 `virtual` 声明。作为一个经验，所有的析构函数都应当用关键字 `virtual` 声明。如果前面的析构函数没有用 `virtual` 声明，代码也可以正常运行。但是，如果代码可能对一个超类指针调用删除操作，但这个超类指针实际上指向一个子类，那么析构链的开始位置就不对了。例如，下述代码与前一个例子是类似的，但是析构函数没有用 `virtual` 声明。当把一个 `Child` 对象作为指向 `Parent` 的指针来访问，然后将其删除时，就会产生问题。

```
class Something
{
public:
    Something() { cout << "2"; }
    ~Something() { cout << "2"; } // Should be virtual, but will work
};
```

```
class Parent
{
public:
    Parent() { cout << "1"; }
    ~Parent() { cout << "1"; } // BUG! Make this virtual!
};
```

```
class Child : public Parent
{
public:
    Child() { cout << "3"; }
    ~Child() { cout << "3"; } // Should be virtual, but will work
```

```
protected:
    Something mDataMember;
};
```

```
int main(int argc, char** argv)
{
    Parent* ptr = new Child();
    delete ptr;
}
```

这段代码的输出结果非常短，就是“1231”。删除变量 `ptr` 时，只调用了 `Parent` 析构函数，因为 `Child` 析构函数没有用 `virtual` 声明。结果没有调用 `Child` 析构函数，其数据成员的析构函数也没有被调用。

从理论上说，只要用 `virtual` 声明 `Parent` 的析构函数就能解决这个问题。“虚拟性”会自动应用到子类上。但是，我们建议将所有的析构函数都用 `virtual` 声明，这样就不必担心这些问题了。

要把所有的析构函数都用 `virtual` 声明。



### 10.3.3 引用父类的数据

在子类中函数和数据成员的名称可能会产生二义性，进行多重继承时尤其如此（参见下文）。C++提供了一种机制来消除两个类之间的名字二义性：作用域解析操作符。它的语法（双冒号）与在类中引用 static 数据是相同的。

在子类中覆盖方法时，实际上是在替换其他代码感兴趣的原始代码。不过父类中的方法依然存在，可能还会用到。但是如果只是通过方法名简单调用，编译器会假定你是想调用子类覆盖的方法。这很容易导致死循环，如下例所示：

```
Sub::doSomething()  
{  
    cout << "In Sub's version of doSomething()" << endl;  
    doSomething(); // BUG! This will recursively call this method!  
}
```

要明确调用父类中的方法，只要在方法前面加上父类名和双冒号即可：

```
Sub::doSomething()  
{  
    cout << "In Sub's version of doSomething()" << endl;  
    Super::doSomething(); // call the parent version.  
}
```

在 C++ 中，调用当前方法的父类实现是常用的一种模式。如果已有一个子类链，那么每个子类可能都要执行超类中已经定义的操作，另外还会额外增加自己的功能。

例如，可以考虑图书类型的类层次结构。该层次结构如图 10-4 所示。

在这个结构中，两个层次较低类都指定了书的类型，我们需要用一个方法来获取书的描述，而它要考虑这个层次体系的所有层次。像前面那样针对父类方法建立方法链可以做到这一点。下面的代码说明了这种模式：

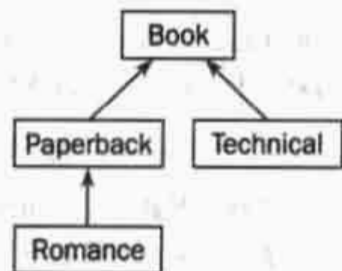


图 10-4

```
#include <iostream>  
#include <string>  
  
using namespace std;  
  
class Book  
{  
public:  
    virtual string getDescription() { return "Book"; }  
};  
  
class Paperback : public Book  
{  
public:  
    virtual string getDescription() {  
        return "Paperback " + Book::getDescription();  
    }  
};
```



```

class Romance : public Paperback
{
public:
    virtual string getDescription() {
        return "Romance " + Paperback::getDescription();
    }
};

class Technical : public Book
{
public:
    virtual string getDescription() {
        return "Technical " + Book::getDescription();
    }
};

int main()
{
    Romance novel;
    Book book;
    cout << novel.getDescription() << endl; // Outputs "Romance Paperback Book"
    cout << book.getDescription() << endl; // Outputs "Book"
}

```

#### 10.3.4 向上类型强制转换和向下类型强制转换

前面已经看到，子类对象可以强制转换为父类或者赋值给父类。如果这种转换或者赋值是在无格式的旧对象上进行，就会造成切割问题。

```
Super mySuper = mySub; // SLICE!
```

像这种情况就会发生切割问题，因为最后的结果是 Super 对象，而 Super 对象没有类 Sub 自己定义的附加功能。但是，如果把子类赋给其超类的指针或者引用，就不会发生切割问题了。

```
Super& mySuper = mySub; // No slice!
```

按超类来引用子类通常是正确的，也称为向上类型强制转换（upcasting）。

该方法和函数引用类比直接使用这些类的对象要好，这就是其中的原因。通过引用，可以对子类进行无切割传递。

进行向上类型强制转换时，要使用指向超类的指针或引用来避免切割问题。

从超类到子类的类型转换也称为向下类型强制转换（downcasting），它常常使专业的 C++ 程序员皱眉头。其中的原因就是没有办法保证对象真正属于该子类。我们来看下面这段代码：

```

void presumptuous(Super* inSuper)
{
    Sub* mySub = static_cast<Sub*>(inSuper);
    // Proceed to access Sub methods on mySub.
}

```

如果编写方法 presumptuous() 的程序员自己来写调用 presumptuous() 的代码，可能就没有什么问题，因为作者自己知道该函数希望参数类型为 Sub\*。

可是，如果另一个程序员要调用 `presumptuous()`，他给该方法传递的参数类型可能是 `Super*`。并不会完成编译时检查来对参数进行类型转换，函数会盲目地假定 `inSuper` 实际上是指向 `Sub` 对象的指针。

向下类型强制转换有时是必须的，在可控的环境中可以有效使用向下类型强制转换。但是如果要进行向下类型强制转换，应该利用 `dynamic_cast`，它使用该类型对象的内置知识来防止无意义的类型转换。

所以前一个例子应该写为如下形式。

```
void lessPresumptuous(Super* inSuper)
{
    Sub* mySub = dynamic_cast<Sub*>(inSuper);
    if (mySub != NULL) {
        // Proceed to access Sub methods on mySub.
    }
}
```

向上面那样，若是对指针不能进行动态类型转换，指针的值为 `NULL`，而不是指向无意义数据。如果未能对对象的引用使用 `dynamic_cast`，则会抛出 `std::bad_cast` 异常。关于类型转换的更多知识，参见第 12 章。关于异常的更多知识参见第 15 章。

只有在必须并且能保证使用动态类型转换的时候才使用向下类型强制转换。

## 10.4 继承以实现多态

现在你已经了解了子类与父类之间的关系，这样就可以在多态中使用继承了，继承在这种情况下的功能是最强大的。第 3 章已经讲到，通过多态性，对象可以和一个公共父类交替使用，也可以使用对象代替父类。

### 10.4.1 Spreadsheet 的返回结果

第 8 章和第 9 章以电子表格应用程序为例介绍了面向对象的程序设计。可以回忆一下，`SpreadsheetCell` 代表单个的数据单元。该元素的类型可以是 `double`，也可以是 `string`。下面定义了一个简化的 `SpreadsheetCell`。注意，其中的单元可以是 `double`，也可以是 `string`。不过在这个例子中，返回的单元格当前值总是 `string` 类型。

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();

    virtual void set(double inDouble);
    virtual void set(const std::string& inString);
    virtual std::string getString();

protected:
    static std::string doubleToString(double inValue);
    static double stringToDouble(const std::string& inString);

    double mValue;
    std::string mString;
};
```

前面的 SpreadsheetCell 类看起来有一个相同的问题——单元格有时表示 double 类型，有时表示 string 类型，有时还要在两者之间进行转换。虽然给定的单元格应该只能存储一个值，但要实现这一点，该类需要存储两个值。更糟的是，是否还需要有其他类型的单元格呢，比如公式单元格或者日期单元格？为了支持所有的数据类型以及这些数据类型之间的转换，类 SpreadsheetCell 将会飞速地增长。

#### 10.4.2 设计多态电子表格单元格

对于这种层次改变，类 SpreadsheetCell 已经不堪重负。一个合理的方法是窄化 SpreadsheetCell 的适用范围，使它仅仅包含 string 类型，可能会在此过程中把它重命名为 StringSpreadsheetCell。为了能够处理 double 类型，可以从 StringSpreadsheetCell 派生另一个类 DoubleSpreadsheetCell，并针对自己的格式提供具体的功能。图 10-5 说明了这一点。这个方法是对使用继承实现重用的建模，因为类 DoubleSpreadsheetCell 只有派生 StringSpreadsheetCell 类，才能利用它的一些内置功能。

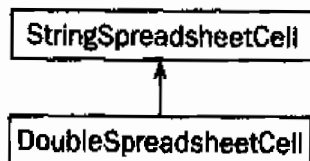


图 10-5

如果要实现图 10-5 的设计，你可能会发现，子类将会覆盖基类的绝大部分（甚至是全部）方法。因为在多数情况下，double 类型的处理与 string 类型是不一样的，它们之间的关系并不像开始理解的那样。而且很明显，包含 string 和包含 double 的单元格之间存在关系。图 10-5 的模型暗含着 DoubleSpreadsheetCell 与 StringSpreadsheetCell 之间的某种“is-a”关系，比起该模型来，一种更好的设计是使这些类对应一个公共的父类 SpreadsheetCell。该设计如图 10-6 所示。

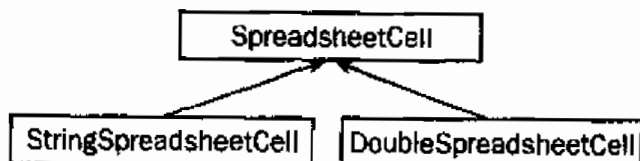


图 10-6

图 10-6 的设计给出了类 SpreadsheetCell 层次结构的一个多态方法。因为类 DoubleSpreadsheetCell 和 StringSpreadsheetCell 是从一个共同的父类 SpreadsheetCell 派生来的，所以从其他代码的角度来看，这两个类是可以互换的。实际上，这意味着：

- 两个子类都支持基类定义的相同的接口（方法集合）。
- 使用 SpreadsheetCell 对象的代码可以调用接口中的任意方法，无须知道单元格是 DoubleSpreadsheetCell 对象还是 StringSpreadsheetCell 对象。
- 通过 virtual 方法的魔力，根据对象的类可以调用接口中的正确方法。
- 其他的数据结构（比如第 9 章定义的类 Spreadsheet）可以通过引用父类型来包含多种类型的单元格集合。

#### 10.4.3 电子表格单元格的基类

电子表格的所有单元格都是基类 SpreadsheetCell 的子类，所以先编写类 SpreadsheetCell 可能会比较好。在设计基类时，需要考虑子类之间的相互关系。根据这个信息，可以先推导出父类应有的共性。例如，string 类型的单元格和 double 类型的单元格都只包含一条数据，在这一点上二者是类似的。由于数据来自于用户，而且还要显示给用户，所以数据值设置为 string 类型并且作为 string 类型获取。这些行为就是构成基类的共有功能。

### 第一次尝试

基类 SpreadsheetCell 的作用是定义所有的子类都支持的行为。在我们简单的例子中，所有的单元格要都能够将值设置为 string 类型。全部单元格还需要能够把当前值作为 string 类型返回。所以，基类的定义中要声明这些方法。

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    virtual ~SpreadsheetCell();

    virtual void set(const std::string& inString);

    virtual std::string getString() const;
};
```

在给该类开始编写 .cpp 文件时，很快会遇到一个问题。由于电子表格单元格的基类既不包含 double 类型也不包含 string 类型，那么如何实现呢？更普遍的问题是，父类要声明子类支持的行为，而又不能实际定义这些行为的实现，那么如何编写父类呢？

一种可能的解决方法就是让这些行为什么都不做。例如，调用基类 SpreadsheetCell 的 set() 方法将没有什么作为，因为基类并不需要设置什么。但是这个方法还不是很好。理想的方法是不应当存在作为基类实例的对象。调用 set() 方法应该总是有作用的，因为它总是在 DoubleSpreadsheetCell 或 StringSpreadsheetCell 上调用。好的解决方案就应该保证这种限制。

### 纯虚方法与抽象基类

纯虚方法 (Pure virtual method) 是指在类定义中显式未定义的方法。把方法设置为纯虚方法，就是告知编译器，在当前类中没有对该方法进行定义。这样的类叫做抽象类 (abstract)，因为没有代码能够对其实例化。编译器可以保证，如果类包含一个或多个纯虚方法，就不能构建此类的对象。

声明纯虚方法的语法如下。在类定义中简单地设置方法等于 0 即可。在 .cpp 文件中不需要编写代码。

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    virtual ~SpreadsheetCell();

    virtual void set(const std::string& inString) = 0;

    virtual std::string getString() const = 0;
};
```

现在基类是抽象类了，所以不可能创建 SpreadsheetCell 对象。下面的代码不能编译，它会报告错误，比如：Cannot declare object of type 'SpreadsheetCell' because one or more virtual functions are abstract. (不能声明 SpreadsheetCell 类型的对象，因为其中的一个或多个函数是抽象函数)。

```
int main(int argc, char** argv)
{
    SpreadsheetCell cell; // BUG! Attempts to create instance of an abstract class
}
```

抽象类提供了一种方法，通过该方法可以阻止其他代码直接实例化此类的对象，这与其子类完全相反（可以实例化子类对象）。

### 基类的源代码

SpreadsheetCell.cpp 并不需要太多的源代码。像定义类一样，大部分的方法都是纯虚方法——不用定义。剩下的工作就是类型转换方法以及构造函数和析构函数。对于这个例子，构造函数和析构函数实现为就像是标志占位符，以备将来可能要做初始化和撤销工作。

```
SpreadsheetCell::SpreadsheetCell()
{
}

SpreadsheetCell::~SpreadsheetCell()
{
}
```

### 10.4.4 各个子类

编写类 StringSpreadsheetCell 和 DoubleSpreadsheetCell 也就是实现父类中定义的功能。因为我们要求客户代码能够实例化 string 和 double 单元格，并且能够共同工作，所以它们不能声明为抽象方法——它们必须实现从父类继承来的纯虚方法。

#### 字符串型电子表格单元格的定义

编写 StringSpreadsheetCell 的类定义的第一步是继承 SpreadsheetCell。

```
class StringSpreadsheetCell : public SpreadsheetCell
{
```

StringSpreadsheetCell 声明自己的构造函数，从而有机会初始化自己的数据。

```
public:
    StringSpreadsheetCell();
```

接着，覆盖所继承的纯虚方法，但是不用设置为 0。

```
virtual void set(const std::string& inString);
virtual std::string getString() const;
```

最后给 string 单元格增加一个 protected 数据成员 mValue，它用来存储实际的单元格数据。

```
protected:
    std::string mValue;
};
```

#### 字符串型电子表格单元格的实现

StringSpreadsheetCell 的 .cpp 文件比起基类的 .cpp 文件来要有趣一些。在构造函数中，值 mValue 初始化为一个 string 类型，表明没有设定值。

```
StringSpreadsheetCell::StringSpreadsheetCell() : mValue("#NOVALUE")
{
}
```



set 方法是直接了当的, 因为内部的表示已经是一个字符串了。类似地, 方法 getString() 只是简单地返回存储的值。

```
void StringSpreadsheetCell::set(const string& inString)
{
    mValue = inString;
}

string StringSpreadsheetCell::getString() const
{
    return mValue;
}
```

### 双精度型电子表格单元格的定义与实现

该类的 double 类型方法也采用了类似的模式, 但是逻辑上有所不同。除采用 string 类型的 set() 方法之外, 它还提供了一个新的 set() 方法, 该方法允许客户代码设置 double 类型的值。还包括两个新的 protected 方法, 用于在 string 和 double 之间转换。像 StringSpreadsheetCell 一样, 它也有一个数据成员 mValue, 但是这是 double 类型的。由于 DoubleSpreadsheetCell 和 StringSpreadsheetCell 是“兄弟”关系, 所以不会发生隐藏和命名冲突。

```
class DoubleSpreadsheetCell : public SpreadsheetCell
{
public:
    DoubleSpreadsheetCell();
    virtual void set(double inDouble);
    virtual void set(const std::string& inString);

    virtual std::string getString() const;

protected:
    static std::string doubleToString(double inValue);
    static double stringToDouble(const std::string& inValue);
    double mValue;
};
```

DoubleSpreadsheetCell 的实现如下:

```
DoubleSpreadsheetCell::DoubleSpreadsheetCell() : mValue(-1)
{
}
```

取 double 的 set() 方法很简单。String 版本的 set() 方法则使用了 protected static 方法 stringToDouble()。方法 getString() 要把存储的 double 类型的值转换为 string 类型:

```
void DoubleSpreadsheetCell::set(double inDouble)
{
    mValue = inDouble;
}

void DoubleSpreadsheetCell::set(const string& inString)
{
    mValue = stringToDouble(inString);
}
```



```

}

string DoubleSpreadsheetCell::getString() const
{
    return doubleToString(mValue);
}

```

你可能已经看到了在层次结构中实现电子表格单元格的主要优点——代码十分简单。不用担心要使用两个字段表示两个数据类型。每个对象都不会受到外界的影响，而且只会处理自己的功能。

需要说明的是，这里省略了 `doubleToString()` 和 `stringToDouble()` 的实现，它们与第 8 章的实现是相同的。

#### 10.4.5 充分利用多态

既然 `SpreadsheetCell` 的层次结构是多态的，所以客户代码可以利用多态性带来的诸多便利。下面的测试程序就探讨了其中的多项特征。

```

int main(int argc, char** argv)
{

```

在此，首先声明一个包含三个 `SpreadsheetCell` 指针的数组。记住，`SpreadsheetCell` 是抽象类，所以不能创建该类型的对象。但是，仍然可以使用指向 `SpreadsheetCell` 的指针或引用，它实际上是指向其中的一个子类。这个数组利用了两个子类之间的公共类型。数组的每个元素可以是 `StringSpreadsheetCell` 类型的，也可以是 `DoubleSpreadsheetCell` 类型的。它们有一个共同的父类，因此可以存储在一起。

```

    SpreadsheetCell* cellArray[3];

```

数组的第 0 个元素设置为指向新的 `StringSpreadsheetCell` 对象的指针，第 1 个元素也设置为指向新的 `StringSpreadsheetCell` 对象的指针，第 2 个则是指向新的 `DoubleSpreadsheetCell` 对象的指针。

```

    cellArray[0] = new StringSpreadsheetCell();
    cellArray[1] = new StringSpreadsheetCell();
    cellArray[2] = new DoubleSpreadsheetCell();

```

现在该数组包含了多种类型的数据，基类声明的任意方法都可以应用到数组对象上。这段代码仅仅使用了 `SpreadsheetCell` 指针——编译器在编译时并不知道对象实际上是什么类型。但是它们都是 `SpreadsheetCell` 的子类，所以肯定都支持 `SpreadsheetCell` 的方法。

```

    cellArray[0]->set("hello");
    cellArray[1]->set("10");
    cellArray[2]->set("18");

```

调用方法 `getString()` 时，每个对象都会正确返回其值的一个 `string` 表示。要意识到重要的一点，这也有些让人惊奇，不同的对象实现的方式也有所不同。`StringSpreadsheetCell` 只是简单返回存储的值，而 `DoubleSpreadsheetCell` 首先执行转换。作为程序员，你不需要知道对象如何操作——只需要知道因为对象是 `SpreadsheetCell` 对象，它能执行该行为即可。

```

    cout << "Array values are [" << cellArray[0]->getString() << ", " <<
        cellArray[1]->getString() << ", " <<
        cellArray[2]->getString() << "]" << endl;
}

```

#### 10.4.6 将来的考虑

从面向对象的程序设计的思想来看,类 SpreadsheetCell 结构的新实现无疑得到了改进。但是由于某些原因,它可能还不足以作为电子表格程序的实际类层次体系。

首先,虽然改进了该结构的设计,但是仍然缺少原来的一个特性:从一种单元格类型转换为另一种类型的能力。把它们分为两个类,单元格对象的集成度就更加松散了。要能够支持从 DoubleSpreadsheetCell 到 StringSpreadsheetCell 的转换,可以增加一个类型构造函数 (typed constructor)。它看起来和复制构造函数很像,但是复制构造函数引用同一个类的对象,而类型构造函数引用的是兄弟类的对象。

```
class StringSpreadsheetCell
{
public:
    StringSpreadsheetCell();
    StringSpreadsheetCell(const DoubleSpreadsheetCell& inDoubleCell);
    ...
}
```

使用类型构造函数,给定 DoubleSpreadsheetCell,可以轻松创建 StringSpreadsheetCell。但是,不要被类型转换所迷惑。从一个兄弟类转换到另一个兄弟类是不行的,除非像第 16 章介绍的那样,重载 (overload) 类型转换操作符。

在一个类型层次体系中,向上类型强制转换总是可以的,向下类型强制转换有些时候是可以的,但是不同层次体系之间进行类型强制转换则是不行的,除非修改了类型强制转换操作符的行为。

如何为单元格实现重载操作符,这个问题比较有趣,可能的解决方法不只一种。一种方法是给单元格的每种组合都实现所有的操作符。只有两个子类的情况下,管理起来比较容易。可以用 operator+ 函数使两个 double 单元格相加、两个 string 单元格相加或者一个 double 单元格和一个 string 单元格相加。另一种方法是确定公共的表示。前一种实现已经做了标准化,即以 string 类型作为各类型的公共表示。一个 operator+ 函数可以利用这个公共表示涵盖所有的情况。一个可能的实现如下所示,这里假定两个单元格相加的结果总是 string 单元格。

```
const StringSpreadsheetCell operator+(const StringSpreadsheetCell &lhs,
                                     const StringSpreadsheetCell &rhs)
{
    StringSpreadsheetCell newCell;
    newCell.set(lhs.getString() + rhs.getString());
    return (newCell);
}
```

只要编译器能够把特定的单元格变成 StringSpreadsheetCell 对象,操作符就能正常执行。对于以上例子,其中的 StringSpreadsheetCell 构造函数以 DoubleSpreadsheetCell 作为参数,如果这是使 operator+ 正常工作的惟一方法,编译器会自动执行转换。这意味着下面的代码可以执行,不过这里明确指定 operator+ 在 StringSpreadsheetCell 上操作。

```
DoubleSpreadsheetCell myDbl;

myDbl.set(8.4);

StringSpreadsheetCell result = myDbl + myDbl;
```

当然，这个加法的结果不是真正地把这些数加在一起。它是把 double 单元格转换为 string 单元格，然后完成 string 相加，结果是一个 SpreadsheetCell 类型的单元格，值为 8.48.4。

如果你对多态性仍然有点不确定，请重新阅读这个例子的代码，明确有关问题。前一个例子中的 main 函数是试验代码的一个不错的起点，其中运用了类的各个方面。

## 10.5 多重继承

在第 3 章已经读到，在面向对象程序设计中，多重继承常常被认为是复杂而且不必要的。它是否有用，我们留给你和你的同事来决定。这一节将解释 C++ 中多重继承的机制。

### 10.5.1 从多个类继承

从语法的角度看，定义有多个父类的类十分简单。需要做的仅仅是声明类名时分别列出各个超类而已。

```
class Baz : public Foo, public Bar
{
    // Etc.
};
```

列出了多个父类之后，Baz 对象就有了如下特性：

- Baz 对象支持 Foo 和 Bar 的所有 public 方法，并包含两个类的所有数据成员。
- 类 Baz 的方法可以访问 Foo 和 Bar 中的 protected 数据和方法。
- Baz 对象可以转换为 Foo 和 Bar 对象。
- 创建新的 Baz 对象时会自动调用 Foo 和 Bar 的默认构造函数，调用顺序为在类定义中这两个类的列出顺序。
- 删除 Baz 对象时会自动调用 Foo 和 Bar 的析构函数，调用顺序与类定义中两个类列出的顺序相反。

下面这个简单的例子给出了一个类 DogBird，它有两个父类——类 Dog 和 Bird。狗鸟是一个荒谬的例子，但是不要认为这说明多重继承本身是荒谬的。说实在的，我们是要让你自己来判断多重继承是否有存在的必要。

```
class Dog
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
};

class Bird
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
};

class DogBird : public Dog, public Bird
{
};
```

类 DogBird 的层次结构如图 10-7 所示。

使用有多个父类的类的对象与使用只有一个父类的类的对象没有什么不同。实际上，客户代码甚至并不知道一个类有两个父类。关键是类支持的属性和行为。在这个例子中，DogBird 的对象支持 Dog、Bird 和所

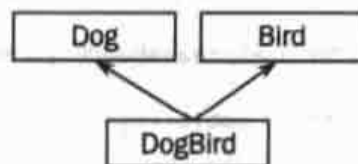


图 10-7

有 public 方法。

```
int main(int argc, char** argv)
{
    DogBird myConfusedAnimal;

    myConfusedAnimal.bark();
    myConfusedAnimal.chirp();
}
```

该程序的输出结果为：

Woof!

Chirp!

### 10.5.2 命名冲突与二义基类

其实构建一个不应当采用多重继承的情况并不难。下面的例子就说明了必须要考虑的一些边缘情况。

#### 名字二义性

如果类 Dog 和 Bird 都有一个 eat() 方法会有什么结果？无论如何 Dog 和 Bird 也不会有关系，所以其中一个 eat() 不会覆盖另一个——在子类 DogBird 中，两种方法同时存在。

只要客户代码永远都不调用 eat() 方法，什么问题都没有。虽然有两个版本的 eat() 方法，但是类 DogBird 可以通过编译。一旦客户代码调用 eat() 方法，编译器就会报告错误，指出调用 eat() 是有二义的。编译器并不知道应该调用哪个 eat()。下面这段代码就会引起这种二义性错误。

```
class Dog
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
    virtual void eat() { cout << "The dog has eaten." << endl; }
};

class Bird
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
    virtual void eat() { cout << "The bird has eaten." << endl; }
};

class DogBird : public Dog, public Bird
{
};

int main(int argc, char** argv)
{
    DogBird myConfusedAnimal;

    myConfusedAnimal.eat(); // BUG! Ambiguous call to method eat()
}
```

要解决二义性问题，或者明确地向上强制转换对象类型，从编译器上隐藏不希望调用的方法，或者使用消除二义性的语法。比如，下面这段代码给出了两种方法调用 Dog 的 eat() 方法。

```
static_cast<Dog>(myConfusedAnimal).eat(); // Slices, calling Dog::eat()
myConfusedAnimal.Dog::eat();             // Calls Dog::eat()
```

通过使用访问父类方法的语法 (`::` 操作符), 子类自己的方法也可以明确消除同名的不同方法之间的二义性。例如, 类 `DogBird` 可以定义自己的 `eat()` 方法来防止代码中的二义性错误。在这个方法中, 要确定调用哪个父类的 `eat()` 方法。

```
void DogBird::eat()
{
    Dog::eat(); // Explicitly call Dog's version of eat()
}
```

引起二义性问题的另一种原因是从同一个类继承两次。比如, 如果类 `Bird` 由于某种原因继承了类 `Dog`, 那么类 `Dog` 就成了具有二义性的基类, 所以类 `DogBird` 的代码不能编译通过。

```
class Dog {};

class Bird : public Dog {};

class DogBird : public Bird, public Dog {}; // BUG! Dog is an ambiguous base
class.
```

产生二义基类时, 多数情况要么是上述这种复杂的 “what-if” 例子, 要么是因为不合适的类层次体系造成的。图 10-8 给出了一个上述例子的类图, 以此说明其中的二义性。

二义性也会发生在数据成员身上。如果类 `Dog` 和 `Bird` 都有一个名字相同的数据成员, 客户代码试图访问该成员时就会产生二义性错误。

### 二义基类

更可能的情况是多个父类自身会有共同的父类。例如, 可能类 `Bird` 和 `Dog` 都是类 `Animal` 的子类, 如图 10-9 所示。

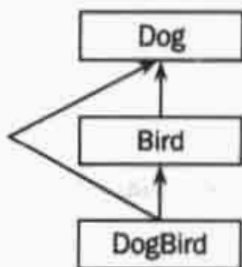


图 10-8

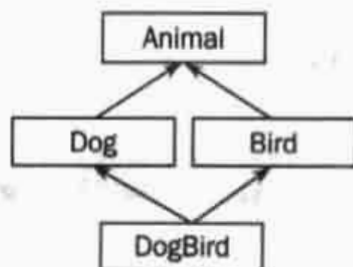


图 10-9

虽然这种类型的类层次结构可能会产生名字的二义性, 但在 C++ 中这种结构是允许的。例如, 若类 `Animal` 有一个 `public` 方法 `sleep()`, 这个方法就不能在 `DogBird` 对象中调用, 编译器并不知道应该调用从 `Dog` 继承来的 `sleep()` 方法还是调用从 `Bird` 继承来的 `sleep()` 方法。

使用菱形的类层次结构的最好方法是使最上层的类成为抽象的基类, 所有的方法都声明成纯虚方法。因为类只声明了方法, 并没有对方法进行定义, 所以在基类中没有可调用的方法, 因此在该层次上不会产生二义性。

下面这个例子使用纯虚 `eat()` 方法实现了菱形类层次结构, 该方法必须在每个子类中定义。类 `DogBird` 仍然要明确规定它使用哪个父类的 `eat()` 方法, 但是不会由于类 `Dog` 和 `Bird` 有同名的方法而引起二义性了, 这并不是因为类 `Dog` 和 `Bird` 都继承自同一个类。



```
class Animal
{
public:
    virtual void eat() = 0;
};

class Dog : public Animal
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
    virtual void eat() { cout << "The dog has eaten." << endl; }
};

class Bird : public Animal
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
    virtual void eat() { cout << "The bird has eaten." << endl; }
};

class DogBird : public Dog, public Bird
{
public:
    virtual void eat() { Dog::eat(); }
};
```

要处理菱形层次结构顶层的类（虚基类），还有一种改进的机制，将在这一章的最后解释。

### 多重继承的用法

读到这里，你可能会感到疑惑，程序员为什么要在程序中使用多重继承呢？使用多重继承的最直观的情况是，要为某种事物的对象定义类，而这个对象同时也是另外一种事物。在第 3 章已经提到，遵循这种模式的现实对象未必能够很好地转换为代码。

必须使用多重继承的简单情况是混合类的实现。混合类已经在第 3 章解释过。使用多重继承来实现的例子将在第 25 章给出。

有时使用多重继承的另一个原因是建立基于组件的类模型。第 3 章给出了一个飞机模拟器的例子。类 Airplane 包括引擎、机身、控制系统和其他组件。Airplane 类的典型实现把每个组件都声明为独立的数据成员，所以可以使用多重继承。飞机类可以继承引擎、机身、控制系统，从而具有所有组件的行为和属性。建议你不要使用这类代码，因为它会使继承与清晰的 has-a 关系相混淆，而继承本应该用于 is-a 关系。

## 10.6 继承技术中有趣而隐蔽的问题

对类进行扩展又引出了各种各样的问题。类的哪些特性可以修改，哪些特性不能修改呢？神秘的关键词 virtual 到底都做了些什么？下面这一节就回答这些问题，这一节会解释其他的许多问题。

### 10.6.1 改变覆盖方法的特性

多数情况下，方法覆盖的原因是为了改变其实现。但是有时是需要改变这些方法的其他特性。

#### 改变方法的返回类型

一个好的经验是使用准确的方法声明，或者叫方法签名（method signature）来覆盖超类使用的方法。方法的实现可以改变，但是签名是相同的。当然，也不必非得这样。在 C++ 中，只要原来的返回类型是指向类的指针或引用，新的返回类型是指向派生类的指针或引用，覆盖的方法就可以改变返回类型。这样的类型称为协变返回类型（covariant returns type）。超类和子类与平行层次结构的对象相互合作时，这种特性有时比较方便。也就是说，另一组类不同于第一个类层次体系，但与之相关。



比如模仿一个樱桃园。其中可以有两个类层次体系，它们对不同的现实对象建模，但是这两个类显然是相关的。首先是 Cherry 链，基类 Cherry 有一个子类 BingCherry。类似地，还有另一个类链，基类为 CherryTree，子类为 BingCherryTree。图 10-10 给出了这两个类链。

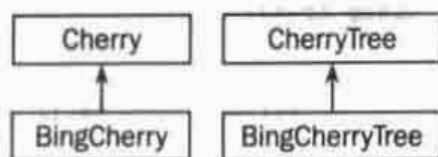


图 10-10

现在假设类 CherryTree 有一个方法 pick()，它的功能是从树上摘一颗樱桃：

```

Cherry* CherryTree::pick()
{
    return new Cherry();
}
  
```

在子类 BingCherryTree 中可能需要覆盖该方法。也许冰樱桃在采摘的时候需要擦亮（在这一点上请原谅我们）。因为 BingCherry 也是 Cherry，所以该方法的签名可以保留，只要像下面的例子一样覆盖方法就行了。BingCherry 指针会自动转换为 Cherry 指针。

```

Cherry* BingCherryTree::pick()
{
    BingCherry* theCherry = new BingCherry();

    theCherry->polish();

    return theCherry;
}
  
```

上述实现十分完美，作者可能就是这样写的。但是你事先知道 BingCherryTree 总是返回 BingCherry 对象，所以可以通过修改返回类型，把这个事实告知该类的潜在用户，如下所示：

```

BingCherry* BingCherryTree::pick()
{
    BingCherry* theCherry = new BingCherry();

    theCherry->polish();

    return theCherry;
}
  
```

确定是否能够改变覆盖方法返回类型有一个好方法：修改之后看现有的代码能否继续运行。在先前的例子中，改变返回类型不会产生什么问题，因为只要代码假定 pick() 方法总是返回 Cherry\*，在修改之后仍然能够通过编译并正确运行。BingCherry 也是 Cherry，所以在 CherryTree 的 pick() 方法的结果上调用的方法也可以在 BingCherryTree 的 pick() 方法的结果上调用。

但是不能把返回类型修改为完全不相关的类型，比如 void\*。下面这段代码不能编译，因为编译器认为你在试图重载 pick() 方法，但是它不能区别 BingCherryTree 的 pick() 方法和 CherryTree 的 pick() 方法，这是因为在消除方法二义性中没有使用返回类型。

```

void* BingCherryTree::pick() // BUG!
{
    BingCherry* theCherry = new BingCherry();
    theCherry->polish();
    return theCherry;
}
  
```

### 改变方法的参数

通常,如果要修改覆盖方法的参数,就不是在覆盖了——因为你创建了一个新方法。返回头来看一下这一章前面使用的 Super 和 Sub 的例子,可以尝试着使用新的参数列表在 Sub 中覆盖方法 someMethod(), 如下所示:

```
class Super
{
    public:
        Super();
        virtual void someMethod();
};
```

```
class Sub : public Super
{
    public:
        Sub();
```

```
        virtual void someMethod(int i); // Compiles, but doesn't override
        virtual void someOtherMethod();
```

```
};
```

该方法的实现如下:

```
void Sub::someMethod(int i)
{
    cout << "This is Sub's version of someMethod with argument " << i
        << "." << endl;
}
```

前一个类定义可以编译通过,但是并没有覆盖方法 someMethod()。因为参数不同,所以你创建了一个新的方法,它只存在于类 Sub 中。如果需要一个 someMethod()方法来操作 int 类型,而且只想操作类 Sub 的对象,那么前一段代码是正确的。子类包含一个与超类中的方法同名的方法,而且相互之间没有实际关系,这样做在风格上是有问题的。

事实上,现在只要涉及 Sub,就会隐藏原来的方法。下面这段代码不能编译通过,因为现在已经没有无参数的 someMethod()方法了。

```
Sub mySub;
```

```
mySub.someMethod(); // BUG! Won't compile because the original method is hidden.
```

还有一种情况是可以修改覆盖方法的参数列表。技巧是新的参数列表必须与原参数列表相兼容。如果修改上面的例子,给参数 i 一个默认值,那么 Sub 的 someMethod()方法实际上就覆盖了 Super 的 someMethod()方法。

```
class Sub : public Super
{
    public:
        Sub();
```

```
        virtual void someMethod(int i = 2); // actually overrides
        virtual void someOtherMethod();
```

这有什么区别吗?当然,覆盖方法的目的是其他的代码要求能够调用该方法,不管是在超类上还是

在子类上，方式应该是相同的。就像前面返回类型的情况一样，要检查能否改变方法参数，需要看是否会修改现有的代码。使用默认参数时，可以调用 Super 的 someMethod() 方法的代码应该就可以调用 Sub 的 someMethod() 方法，而无须任何修改。

下面这段代码说明了在这种情况下，Sub 实际上覆盖了 Super 的 someMethod() 方法。甚至 Super 引用会正确地调用 Sub 的 someMethod() 方法。

```
Sub mySub;
Super& ref = mySub;

mySub.someMethod();    // Calls Sub's someMethod with default argument
mySub.someMethod(1);   // Calls Sub's someMethod
ref.someMethod();      // Calls Sub's someMethod with default argument
```

这段代码的输出结果如下：

```
This is Sub's version of someMethod with argument 2.
This is Sub's version of someMethod with argument 1.
This is Sub's version of someMethod with argument 2.
```

还可以用一种稍微难懂的技术来满足需求。可以使用新的签名来有效覆盖子类的方法，而同时仍然继续继承超类的方法。该技术使用关键字 using 在子类中明确地包含方法在超类中的定义，如下所示：

```
class Super
{
public:
    Super();
    virtual void someMethod();
};

class Sub : public Super
{
public:
    Sub();

    using Super::someMethod; // Explicitly "inherits" the Super version
    virtual void someMethod(int i); // Adds a new version of someMethod
    virtual void someOtherMethod();
};
```

覆盖的方法一般很少改变参数列表。

### 10.6.2 覆盖方法的特殊情况

在覆盖方法时，有一些边缘情况需要特别注意。这一节将概要介绍可能会遇到的情况。

#### 超类方法是 Static 方法

在 C++ 中，不能覆盖 static 方法。大部分情况下知道这一点就行了。不过，还有几点需要了解。

首先，不能同时用 static 和 virtual 声明一个方法。这也说明了试图覆盖 static 方法的结果并不像原想的那样。如果在子类中有一个 static 方法，名字与超类中的 static 方法相同，实际上就有了两个相互独立的方法。

下面这段代码中有两个类，这两个类恰好都有一个 static 方法 beStatic()。但这两个方法是绝对不相关的。

```

class SuperStatic
{
    public:
        static void beStatic() { cout << "SuperStatic being static, yo." << endl;
};

class SubStatic
{
    public:
        static void beStatic() { cout << "SubStatic keepin' it static." << endl;
};

```

因为 static 方法属于自己的类，所以在两个不同的类上调用名字完全相同的方法将会分别调用这两个类自己的方法：

```

SuperStatic::beStatic();
SubStatic::beStatic();

```

这段代码将输出：

```

SuperStatic being static, yo.
SubStatic keepin' it static.

```

只要由类访问这些方法，则一切都很合理。但如果涉及对象的时候，其行为就变得有点不那么清楚了。在 C++ 中，尽管从语法上说，可以在对象上调用 static 方法，但实际上 static 方法只存在于类中。看一下下面这段代码：

```

SubStatic mySubStatic;
SuperStatic& ref = mySubStatic;

mySubStatic.beStatic();
ref.beStatic();

```

很明显，第一次调用 beStatic() 将会调用 SubStatic 的 beStatic 方法，因为它明确指出是在声明为 SubStatic 的对象上调用的。第二个调用就不那么明确了。该对象是 SuperStatic 的引用，但是它指向 SubStatic。在这种情况下，将会调用 SuperStatic 的 beStatic() 方法。原因就是在 C++ 中调用 static 方法时，它并不关心对象实际是什么，而只关心对象编译时的类型。在这种情况下，编译时类型是对 SuperStatic 的引用。

这个例子的输出结果为：

```

SubStatic keepin' it static.
SuperStatic being static, yo.

```

static 方法限于定义该方法的类，而非对象。类中如果有方法调用 static 方法，它们调用的则是在该类中定义的 static 方法版本，这独立于调用原方法的对象的运行时类型。

### 重载超类的方法

覆盖方法时，其实是隐式地隐藏了该方法的其他实现。为什么想要改变某个方法的一些版本，而不改变其他版本呢？想到这一点就想到点上了。看一下下面的这个子类，它覆盖了一个方法，但是没有覆盖相关联的重载方法：

```

class Foo
{
public:
    virtual void overload() { cout << "Foo's overload()" << endl; }
    virtual void overload(int i) { cout << "Foo's overload(int i)" << endl; }
};

class Bar : public Foo
{
public:
    virtual void overload() { cout << "Bar's overload()" << endl; }
};

```

如果试图在 Bar 对象上调用参数类型为 int 的 overload() 方法，代码不能编译通过，因为没有明确覆盖该方法。

```
myBar.overload(2); // BUG! No matching method for overload(int).
```

但是，从 Bar 对象访问这个版本的方法是可以的。所需要的只是指向 Foo 对象的一个指针或者引用。

```

Bar myBar;
Foo* ptr = &myBar;

ptr->overload(7);

```

隐藏未实现的重载方法仅仅是 C++ 的表面。显式声明为子类型实例的对象不会使这些方法可用，但是通过简单的类型强制转换把它转换为超类可以实现这一点。

如果只是想改变方法的一个版本，可以使用关键字 using 来减少重载所有版本的麻烦。在下面这段代码中，类 Bar 的定义使用了 Foo 的 overload() 方法，并显示重载了 overload() 方法的其他版本。

```

class Foo
{
public:
    virtual void overload() { cout << "Foo's overload()" << endl; }
    virtual void overload(int i) { cout << "Foo's overload(int i)" << endl; }
};

class Bar : public Foo
{
public:
    using Foo::overload;
    virtual void overload() { cout << "Bar's overload()" << endl; }
};

```

为了避免不明确的 bug，应该明确指出或者使用关键字 using 来覆盖重载方法的所有版本。

### 超类方法为 Private 或 Protected 方法

覆盖 private 或 protected 方法是完全正确的。记住一点，方法的访问限定符决定了谁可以调用该方法。子类不能调用其父类的 private 方法，但仅凭这一点并不是说它不能覆盖这些函数。实际上在面向对象的语言中，覆盖 private 或者 protected 方法是一种普遍的模式。它允许子类定义它们自己在超类中引用的“特有性”。

比如，下面的类用来模拟汽车器的一部分，它根据汽车的百公里耗油量和剩余的油估算汽车可以行



驶的距离。

```
class MilesEstimator
{
public:
    virtual int getMilesLeft() {
        return (getMilesPerGallon() * getGallonsLeft());
    }

    virtual void setGallonsLeft(int inValue) { mGallonsLeft = inValue; }
    virtual int getGallonsLeft() { return mGallonsLeft; }
private:
    int mGallonsLeft;
    virtual int getMilesPerGallon() { return 20; }
}
```

方法 `getMilesLeft()` 根据它自己的两个方法进行计算。下面这段代码使用 `MilesEstimator` 来计算两加仑汽油可以行驶多少公里。

```
MilesEstimator myMilesEstimator;

myMilesEstimator.setGallonsLeft(2);
cout << "I can go " << myMilesEstimator.getMilesLeft() << " more miles." << endl;
```

输出结果为：

```
I can go 40 more miles.
```

要使得这个类更加有趣，可能还要增加几种不同类型的车辆，如效率更高的车。现有的类 `MilesEstimator` 假定所有的汽车每加仑汽油可以行驶 20 英里，但是这个值是从单独的方法返回的，所以子类可以覆盖它。这样的一个子类如下所示。

```
class EfficientCarMilesEstimator : public MilesEstimator
{
private:
    virtual int getMilesPerGallon() { return 35; }
};
```

通过覆盖这个 `private` 方法，新类完全改变了现有未修改 `public` 方法的行为。超类中的 `getMilesLeft()` 方法会自动调用 `private` `getMilesPerGallon()` 方法覆盖的版本。使用这个新类的例子如下：

```
EfficientCarMilesEstimator myEstimator;

myEstimator.setGallonsLeft(2);
cout << "I can go " << myEstimator.getMilesLeft() << " more miles." << endl;
```

这一次，输出结果反映了覆盖的功能：

```
I can go 70 more miles.
```

要改变类的某些特征而不用大幅调整，对此覆盖 `private` 和 `protected` 方法是一种很好的方法。

### 超类方法有默认参数

子类和超类都可以有不同的默认参数，但是使用的参数依赖于变量的声明类型，而不是底层的对象。



下面是子类的一个简单例子，它在覆盖的方法中提供了不同的默认参数：

```
class Foo
{
public:
    virtual void go(int i = 2) { cout << "Foo's go with param " << i << endl; }
};

class Bar : public Foo
{
public:
    virtual void go(int i = 7) { cout << "Bar's go with param " << i << endl; }
};
```

如果在 Bar 对象上调用 go()，Bar 的 go() 方法将使用默认的参数 7 来执行。如果在 Foo 对象上调用 go()，Foo 的 go() 方法将会使用默认参数 2 来执行。然而（正是这里有些奇怪），如果在实际指向 Bar 对象的 Foo 指针或者 Foo 引用上调用 go()，将会调用 Bar 的 go() 方法，但使用的却是 Foo 的默认参数 2。这个行为如下所示：

```
Foo myFoo;
Bar myBar;
Foo& myFooReferenceToBar;

myFoo.go();
myBar.go();
myFooReferenceToBar.go();
```

这段代码的输出结果如下：

```
Foo's go with param 2
Bar's go with param 7
Bar's go with param 2
```

比较棘手吧，是不是？造成这种行为的原因是 C++ 把默认参数与指示对象的变量类型绑在一起了，而不是与对象本身绑定。也是由于这个原因，在 C++ 中，不能继承默认参数。如果上面的类 Bar 没有像其父类一样提供默认参数，它将重载一个新的 0 参数版本的 go() 方法。

覆盖有默认参数的方法时，也应该提供默认参数，而且往往要有相同的值。

### 超类方法有不同的访问级别

你可能希望改变方法的访问级别，对此有两种方式，可以使之更限定，也可以使限定更弱。哪一种方法在 C++ 中都没有太大的意义，但是关于为什么要这样做有几条理由。

要给方法（或者数据成员，原因是相同的）实施更加严格的约束，可以采取两种方法。一种是改变整个基类的访问限定符。这种方法在这一章的最后介绍。另一种方法在子类中重新定义访问限定符，像下面的这个类 Shy 一样：

```
class Gregarious
{
public:
    virtual void talk() { cout << "Gregarious says hi!" << endl; }
};
```

```
class Shy : public Gregarious
{
    protected:
        virtual void talk() { cout << "Shy reluctantly says hello." << endl; }
};
```

类 Shy 中的 talk() 方法（访问级别为 protected）正确地覆盖了该方法。试图在 Shy 对象上调用 talk() 的任意客户代码都会得到一个编译错误：

```
myShy.talk(); // BUG! Attempt to access protected method.
```

但是，这个方法并不是完全受保护的。只需得到一个 Gregarious 引用或指针就可以访问你认为得到保护的方法：

```
Shy myShy;
Gregarious& ref = myShy;

ref.talk();
```

前面这段代码的输出结果为：

```
Shy reluctantly says hello.
```

这证明在子类中把方法声明为 protected，这确实是覆盖了方法（因为可以正确调用子类中的方法），但是也证明，如果超类使得其为 public，就不能完全保证 protected 访问。

并没有适当的方式（或者合理的原因）来限制访问 public 父方法。

在子类中减少访问约束则容易的多（也更有意义）。最简单的方法就是仅提供一个 public 方法，由它调用超类的一个 protected 方法，如下所示：

```
class Secret
{
    protected:
        virtual void dontTell() { cout << "I'll never tell." << endl; }
};

class Blabber : public Secret
{
    public:
        virtual void tell() { dontTell(); }
};
```

调用 Blabber 对象的 public 方法 tell() 的客户代码可以成功访问类 Secret 的 protected 方法。当然，这并不会真正改变 dontTell() 的访问权限，只是提供了访问它的一个 public 方式。

也可以在子类 Blabber 中明确覆盖 dontTell()，并给它一个具有 public 访问权限的新行为。这比降低访问级别更有意义，因为它十分清楚对基类的引用或指针所发生的事情。例如，假定 Blabber 实际上使 dontTell() 方法成为了 public 方法：

```
class Secret
{
    protected:
        virtual void dontTell() { cout << "I'll never tell." << endl; }
};
```

```
class Blabber : public Secret
{
public:
    virtual void dontTell() { cout << "I'll tell all!" << endl; }
};
```

如果在 Blabber 对象上调用 dontTell() 方法，将输出 I'll tell all!

```
myBlabber.dontTell(); // Outputs "I'll tell all!"
```

然而在这种情况下，超类中的 protected 方法仍然是 protected 方法，因为想通过指针或引用调用 Secret 的 dontTell() 方法的任何企图都不能编译通过。

```
Blabber myBlabber;
Secret& ref = myBlabber;
Secret* ptr = &myBlabber;

ref.dontTell(); // BUG! Attempt to access protected method.
ptr->dontTell(); // BUG! Attempt to access protected method.
```

改变方法访问级别的惟一有用的方式是给 protected 方法提供约束更少的访问方法。

### 10.6.3 复制构造函数与相等操作符

在第 9 章我们讲到，在类中动态分配内存时，复制构造函数和赋值操作符被认为是很好的程序设计实践。在定义子类时，需要小心使用复制构造函数和 operator=。

如果子类没有特殊的数据（通常是指针）需要非默认的复制构造函数或 operator=，你就不需要有非默认的复制构造函数了，不管超类中是否包括非默认的复制构造函数。如果子类省略了复制构造函数，在复制对象时，依然会调用父类的复制构造函数。类似地，如果你没有提供明确的 operator=，将使用默认的赋值操作符，也就是在父类上调用 operator=。

另一方面，如果确实在子类中指定了复制构造函数，需要明确地把它链接到父类的复制构造函数上，如下面这段代码所示。倘若没有这样做，该对象的父类部分就会使用默认的构造函数（而不是复制构造函数）。

```
class Super
{
public:
    Super();
    Super(const Super& inSuper);
};

class Sub : public Super
{
public:
    Sub();
    Sub(const Sub& inSub);
};

Sub::Sub(const Sub& inSub) : Super(inSub)
{
}
```

类似地，如果子类覆盖了操作符=，它一般也总是需要调用父类中的 operator=。不需要这样做的惟一情况是：发生赋值时，由于某种奇怪的原因，仅仅需要给对象的一部分赋值。下面这段代码就说明了如何从子类调用父类的赋值操作符。

```
Sub& Sub::operator=(const Sub& inSub)
{
    if (&inSub == this) {
        return *this;
    }

    Super::operator=(inSub) // Call parent's operator=.

    // Do necessary assignments for subclass.

    return (*this);
}
```

如果子类没有指定复制构造函数或 operator=，父类的功能继续有效。如果子类没有覆盖自己的复制构造函数或 operator=，则需要明确引用父类的复制构造函数或 operator=。

#### 10.6.4 关键字 virtual 的真相

前面第一次出现方法覆盖时，就已经说过，只有 virtual 方法可以正确覆盖。不得不适当地加一个限定符的原因是，如果没有用 virtual 声明方法，你也可以覆盖该方法，但是在某些微妙的情况下，会发生错误。

##### 隐藏而非覆盖

下面这段代码定义了一个超类和一个子类，各自都只有一个方法。子类试图覆盖超类中的方法，但是超类中的方法没有用 virtual 声明。

```
class Super
{
public:
    void go() { cout << "go() called on Super" << endl; }
};

class Sub : public Super
{
public:
    void go() { cout << "go() called on Sub" << endl; }
};
```

试图在 Sub 对象上调用方法 go() 最初看起来是正确的。

```
Sub mySub;

mySub.go();
```

正像你所期待的那样，这个调用的输出结果是 go() called on Sub。但是因为该方法不是 virtual 方法，实际上并没有覆盖它。而是类 Sub 创建了一个新方法，名字也是 go()，它与类 Super 中的 go() 方法没有任何关系。要证明这一点，只要在 Super 指针或引用的环境中简单调用该方法即可。

```

Sub mySub;

Super& ref = mySub;

ref.go();

```

你认为输出结果应该是 go() called on Sub, 但实际上输出结果为 go() called on Super。这是因为变量 ref 是一个 Super 引用, 而且忽略了关键字 virtual。调用方法 go() 时, 它只是简单执行了 Super 的 go() 方法。因为这个方法不是 virtual 方法, 所以不需要考虑是否在子类中覆盖了该方法。

试图覆盖非 virtual 方法将“隐藏”超类的定义, 而且只能在子类的环境中使用。

### virtual 是如何实现的

要了解为什么会发生方法隐藏, 需要多了解一点关键字 virtual 实际都做了哪些工作。在 C++ 中编译类时, 要创建一个二进制对象, 它包含了该类的所有数据成员和方法。在没有关键字 virtual 的情况下, 在基于编译时类型调用的方法处, 跳转到正确方法的代码是直接硬编码 (hard-coded) 的。

如果声明方法时使用了 virtual, 则是在一块特别内存区域寻找方法的实现, 这块内存区域叫做 *vtable*, 就是“虚拟表”的意思。包含一个或多个 virtual 方法的类都有一个 vtable, 它包含了指向 virtual 方法实现的指针。通过这种方式, 在一个对象上调用方法时, 指针就进了 vtable, 并且是基于对象的类型执行该方法的正确版本, 而不是基于用来访问该方法的变量类型来执行。

图 10-11 给出了一个高级视图, 展示了 vtable 如何使得方法覆盖成为可能。该图包括两个类, Super 和 Sub。Super 声明了两个方法, foo() 和 bar()。通过 Super 的 vtable 可以看出, 每个方法都有自己的由类 Super 定义的实现。类 Sub 没有覆盖 Super 的 foo() 方法, 所以 Sub 的 vtable 指向 foo() 的同一个实现。但是 Sub 覆盖了 bar(), 所以它的 vtable 指向新的 bar() 实现。

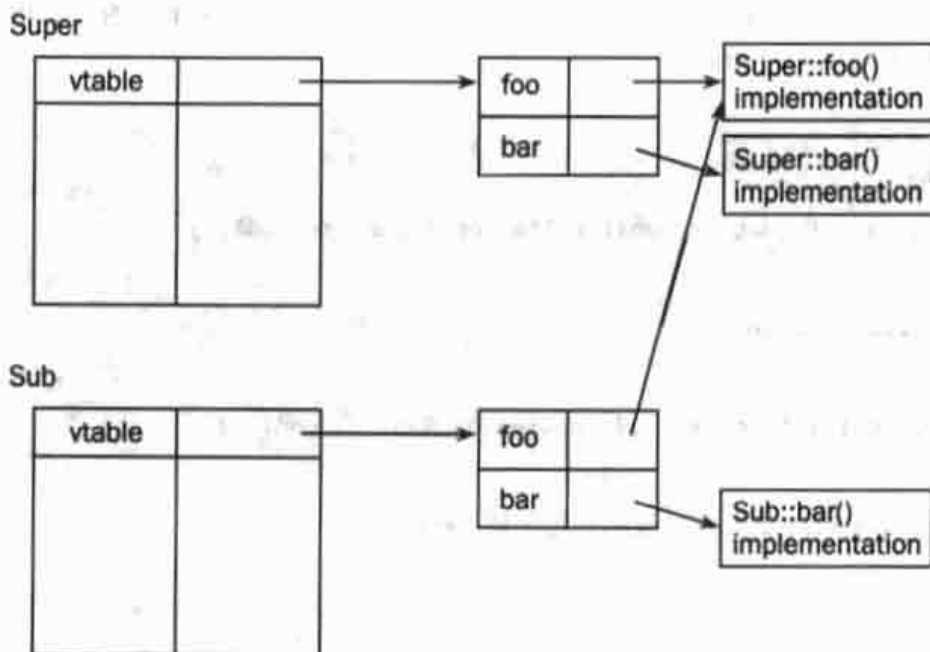


图 10-11

### virtual 的合理性

假如事实是建议所有的方法都声明成 virtual 方法, 你可能就会有疑问了, 为什么还要有关键字 virtual



存在。为什么不能让编译器自动地把所有的方法都设成 virtual 方法呢？可以让编译器这样做。有很多人认为 C++ 语言就应该把所有的都设置为 virtual 的。Java 语言实际上就是这样做的。

之所以会有反对的意见，不支持把一切都声明为 virtual，而认为要创建关键字 virtual，这都与 vtable 开销有关。要调用 virtual 方法，程序就需要对指向待执行代码的指针解除引用，这增加了额外的操作。在多数情况下这会对性能有所影响，但是 C++ 的设计者认为最好让程序员自己决定是否有必要引用这些性能影响，起码到现在是这样。如果方法永远不会被覆盖，就无须把它声明成 virtual 以避免影响性能。这对代码的大小也有一点影响。除了方法的实现之外，每个对象还需要一个指针，它会占用有限的内存空间。

### 非虚析构函数带来的麻烦

即使程序员不采用这个原则，即不会把所有的方法都声明成 virtual 方法，但是对于析构函数他们仍然会坚持该原则。这是因为把析构函数设置为非 virtual 方法很容易引起内存泄漏。

例如，如果子类使用在构造函数中动态分配并在析构函数中删除的内存，若不调用析构函数，那么永远都不会释放这段内存。正如下面这段代码所示，如果析构函数是非 virtual 方法，它很容易欺骗编译器，让它跳过调用析构函数。

```
class Super
{
public:
    Super();
    ~Super();
};

class Sub : public Super
{
public:
    Sub() { mString = new char[30]; }
    ~Sub() { delete[] mString; }

protected:
    char* mString;
};

int main(int argc, char** argv)
{
    Super* ptr = new Sub(); // mString is allocated here.

    delete ptr; // ~Super is called, but not ~Sub because the destructor
               // is not virtual!
}
```

我们强烈建议把所有的方法（构造函数除外）都声明成 virtual 方法，除非有充分的原因不这样做。构造函数不能也不需要声明为 virtual 方法，因为创建对象时，总是会指定要构造的具体类。

### 10.6.5 运行时类型工具

相对于其他的面向对象的程序设计语言，C++ 很大程度上是面向编译时的。像前面学到的那样，覆盖之所以有效，是因为方法及其实现之间有一个间接层，而不是因为对象有类的内置知识。

但是 C++ 有一些特性确实提供了对象的运行时视图。这些特性通常归为一个特征集合，这个特性集



合叫做运行时类型识别 (Runtime Type Identification)，或者叫 RTTI。RTTI 提供了大量有用的特征来处理有关对象类成员的信息。

#### dynamic\_cast

回顾第 1 章，我们曾经介绍过 static\_cast，这是 C++ 的类型转换机制之一。static\_cast 操作符之所以如此命名，是因为转换已经内置于已编译代码中。不管对象的运行时类型是什么，static 的向下类型强制转换总是成功的。

前面向下类型强制转换一节中已经讲到，在 OO 层次结构中，dynamic\_cast 为类型之间的转换提供了一种比较安全的机制。对象的动态类型强制转换的语法类似于 static 类型转换的语法。然而，在动态类型强制转换中，对于指针，无效的转换将返回 NULL，对于引用则抛出异常。下面这个例子说明了如何对引用正确执行动态类型强制转换。

```
SomeObject myObject = getSomeObject();

try {
    SomeOtherObject& myRef = dynamic_cast<SomeOtherObject&>(myObject);
} catch (std::bad_cast) {
    cerr << "Could not convert the object into the desired type." << endl;
}
```

#### typeid

操作符 typeid 可以用来在运行时查询对象找出其类型。大部分情况下无须使用 typeid，因为基于对象类型有条件运行的代码用 virtual 方法处理更好。下面这段代码根据对象类型使用 typeid 来打印消息。

```
#include <typeinfo>

void speak(const Animal& inAnimal)
{
    if (typeid(inAnimal) == typeid(Dog&)) {
        cout << "Woof!" << endl;
    } else if (typeid(inAnimal) == typeid(Bird&)) {
        cout << "Chirp!" << endl;
    }
}
```

只要看到类似的代码，就应该立刻想到应当把该功能重新实现为 virtual 方法。在这种情况下，较好的实现是在类 Animal 中声明一个 virtual 方法 speak()。类 Dog 会覆盖该方法打印“Woof!”，类 Bird 会覆盖该方法打印“Chirp!”。这种方法更能满足面向对象的程序设计，与对象相关的功能都会提供给对象。

但是在调试过程中，typeid 的功能有时很方便。为了进行日志记录和调试，打印出对象的类型是很有帮助的。下面这段代码利用了 typeid 来进行日志记录。函数 logObject 以 loggable 对象作为参数。如果采用这样的设计，可以记录的任何对象就都是类 Loggable 的子类了，并且支持方法 getLogMessage()。通过这种方法，Loggable 就是一个混合类。

```
#include <typeinfo>

void logObject(Loggable& inLoggableObject)
{
    logfile << typeid(inLoggableObject).name() << " ";
    logfile << inLoggableObject.getLogMessage() << endl;
}
```

函数 `logObject()` 首先把对象的类名写入文件中，后面跟着日志信息。以后再读日志的时候，就可以明白哪个对象对应文件的哪一行了。

### 10.6.6 非公共继承

在前面所有的例子中，父类总是使用关键字 `public` 列出。你可能会有疑问，如果父类是 `private` 或 `protected` 的会怎样呢。虽然不像 `public` 那样普遍，实际上这也是可以的。

把子类与父类的关系声明成 `protected`，表示超类的 `public` 和 `protected` 方法与数据成员在子类环境中就都变成了 `protected` 的。类似地，使用 `private` 就等于在子类中把超类的所有方法和数据成员都变成了 `private` 的。

有一些原因可以说明为什么可能需要以这种方式一律降低父类的访问权限，但是大部分原因在层次结构的设计中都隐藏着一些瑕疵。有些程序员会滥用这个语言的特性，常常会把多重继承结合起来，来实现类的“组件”。他们会让类 `Airplane` 包含 `protected` 引擎和 `protected` 机身，而不是包含一个引擎数据成员和一个机身数据成员。这样做的话，对于客户代码，`Airplane` 看起来并不像引擎和机身（因为所有的成员都是 `protected`），但是可以在内部使用所有这些功能。

非 `public` 继承是很少见的，我们建议慎重使用，即便没有其他原因，只是因为大部分程序员不熟悉它，也应该如此。

### 10.6.7 虚基类

本章的前面你已经了解了二义基类，这是当多个父类有一个公共基类时会引起的一种情况，如图 10-9 所示。我们推荐的解决方法是确保共享的父类没有自己的任何功能。这样就永远不会调用它的方法，也就没有二义性问题了。

需要共享的父类有自己的功能时，C++ 还有另外一种机制来解决这个问题。如果共享的父类是虚基类（virtual base class），就不会有二义性了。下面的代码把方法 `sleep()` 加到了基类 `Animal` 中，并修改了类 `Dog` 和 `Bird`，使这两个类以 `Animal` 作为虚基类从 `Animal` 继承。没有关键字 `virtual` 的话，在 `DogBird` 对象上调用 `sleep()` 会引起二义性，因为 `Dog` 和 `Bird` 都从 `Animal` 继承了方法 `sleep()`。然而，虚继承 `Animal` 时，在子孙中每个方法或成员则只有一个副本存在。

```
class Animal
{
    public:
        virtual void eat() = 0;
        virtual void sleep() { cout << "zzzzz...." << endl; }
};
class Dog : public virtual Animal
{
    public:
        virtual void bark() { cout << "Woof!" << endl; }
        virtual void eat() { cout << "The dog has eaten." << endl; }
};

class Bird : public virtual Animal
{
    public:
        virtual void chirp() { cout << "Chirp!" << endl; }
        virtual void eat() { cout << "The bird has eaten." << endl; }
```

```
};

class DogBird : public Dog, public Bird
{
public:
    virtual void eat() { Dog::eat(); }
};

int main(int argc, char** argv)
{
    DogBird myConfusedAnimal;

    myConfusedAnimal.sleep(); // Not ambiguous because Animal is virtual
}
```

在类继承中，虚基类是避免二义性的一个好方法。惟一的缺点是很多 C++ 程序员并不熟悉这个概念。

## 10.7 小结

在本章中，我们带着你学习了继承的许多知识点。你已经学到了它的许多应用，包括代码重用和多态性。还学习了很多滥用的情况，包括设计蹩脚的多重继承方案。沿着这条路，你会发现一些不常见的边缘情况，在日常的基本应用中，它们出现的可能性不大，但是会引起一些糟糕的 bug（和接口问题）。

继承是一个强有力的语言特性，要熟悉它需要一些时间。在学习了本章的例子并自己亲身实验之后，我们希望继承能够变成你在面向对象设计中所选用的工具。

# 第 11 章 利用模板编写通用代码

C++ 不仅支持面向对象的程序设计，同时也支持通用程序设计（generic programming）。在第 5 章我们已经讨论过，通用程序设计的目的是编写可重用的代码。C++ 中通用程序设计的主要工具是模板。虽然严格意义上说，模板不算是面向对象程序设计的特性，但是可以把模板和面向对象程序设计结合起来得到非常强大的功能。遗憾的是，许多程序员认为，模板是 C++ 中最难的部分，出于这个原因，他们往往尽量避免使用模板。然而，要使用 C++ 的标准库，即使你自己从来没有写过模板，也应该了解模板的语法和功能。

本章给出了详细的代码来实现第 5 章讨论的通用性设计原则，并假定你已经对 C++ 标准模板库有所了解，标准模板库的详细内容在第 21 章～23 章讨论。本章分为两部分。第一部分介绍了最常使用的模板特性，包括：

- 如何编写模板类
- 编译器如何处理模板
- 如何组织模板的源代码
- 如何使用无类型模板参数
- 如何编写单个类方法的模板
- 如何针对具体类型定制类模板
- 如何结合模板与继承
- 如何编写函数模板
- 如何使模板函数成为模板类的友元函数

第二部分深入研究了更难懂的模板特性，包括：

- 三种模板参数及其有关细节
- 部分特殊化
- 函数模板的推导
- 如何利用模板递归

## 11.1 模板概述

采用过程式模式，主要程序设计单元是过程（procedure）或函数（function）。函数的用处主要在于，使用函数可以编写独立于特定值的算法，因此可以重复用于多个不同的值。比如，如果程序调用了 C 和 C++ 的函数 `sqrt()`，这个函数就可以计算程序所提供参数的平方根。如果函数只是用来计算一个数（比如 4）的平方根，这种函数并没有多大的用处。不过 `sqrt()` 函数是针对参数（parameter）编写的，该参数可以替换为调用该函数的程序所传递的任何值。计算机科学家称，函数是对值的参数化（parameterize）。

面向对象的程序设计增加了对对象的概念，对象把相关的数据和行为组成为一个整体，但是对象没有

改变函数和方法对值的参数化方式。

模板扩展了参数化的概念，使用模板可以对数据类型（type）以及值（value）参数化。回忆一下 C++ 中的数据类型，它包括基本的数据类型，比如 int 和 double，以及用户定义的类，比如 SpreadsheetCells 和 CherryTrees。使用模板编写的代码可以独立于传递给代码的值，而且独立于这些值的数据类型。例如，为了存储 int、Car 和 SpreadsheetCell 等数据类型，要编写各自的栈类，为此可以编写一个栈类定义，用于存储上述任意一种类型，而不是分别为各个类型编写独立的栈类。

虽然说模板是一个令人惊异的语言特性，但是 C++ 的模板无论在概念上还是在语法上都很让人困惑，使得许多程序员只能敬而远之。C++ 中提供了对模板的支持，这是一个委员会设计的，有时候看起来这个委员会好像采取了一种“统统扔到厨房水池”的混杂方法：许多模板特性的目的性并不是很明确。更糟糕的是，支持模板的编译器从一开始就良莠不齐，而且将来也不会好到哪里。几乎没有哪个商业编译器能够根据 C++ 标准提供对模板的完全支持。

由于这些原因，大部分的 C++ 书籍只是浅显地介绍了模板的表面知识。然而，理解 C++ 模板至关重要，主要是因为 C++ 标准模板库是用模板建立的，要充分利用 C++ 标准模板库，就必须懂得模板的基础知识。

因此，本章将讲解 C++ 对模板的支持，重点是标准模板库中涉及的方面。沿着这条路，你将学到一些极好的特性，除了使用标准模板库之外，还可以在编写程序时使用这些特性。

## 11.2 类模板

类模板（class template）主要用作存储对象的容器或数据结构。本节将使用一个可以实际运行的例子——Grid 容器。为了保证例子代码的篇幅不至于过大，也为了只是说明具体的知识点，本章将分小节为 Grid 容器分别加入一些特性，而各个小节增加的特性只在相应的小节中使用。

### 11.2.1 编写类模板

假定需要设计一个通用的游戏棋盘类，它可以作为国际象棋棋盘、跳棋棋盘、Tic-Tac-Toe（连珠游戏）棋盘或者其他任何一种二维游戏棋盘。针对通用目的，这个类应该能够存储国际象棋棋子、跳棋棋子、连珠游戏棋子或者其他游戏的棋子。

#### 不使用模板编写代码

不使用模板的话，建立通用游戏棋盘的最好方法就是采用多态来存储通用的 GamePiece 对象。然后从类 GamePiece 为每种棋子派生一个子类。例如，在国际象棋游戏中，类 ChessPiece 就是 GamePiece 的子类。通过多态，用来存储 GamePiece 的类 GameBoard 也可以存储 ChessPiece。类定义看起来可能有点类似于第 9 章的 ChessPiece 类，它使用动态分配的二维数组作为底层网格结构：

```
// GameBoard.h

class GameBoard
{
public:
    // The general-purpose GameBoard allows the user to specify its dimensions
    GameBoard(int inWidth = kDefaultWidth, int inHeight = kDefaultHeight);
    GameBoard(const GameBoard& src); // Copy constructor
    ~GameBoard();
    GameBoard &operator=(const GameBoard& rhs); // Assignment operator

    void setPieceAt(int x, int y, const GamePiece& inPiece);
```

```

GamePiece& getPieceAt(int x, int y);
const GamePiece& getPieceAt(int x, int y) const;

int getHeight() const { return mHeight; }
int getWidth() const { return mWidth; }
static const int kDefaultWidth = 10;
static const int kDefaultHeight = 10;

protected:
    void copyFrom(const GameBoard& src);
    // Objects dynamically allocate space for the game pieces.
    GamePiece** mCells;
    int mWidth, mHeight;
};

```

函数 `getPieceAt()` 返回某个指定点上棋子的引用，而不是该棋子的副本。类 `GameBoard` 作为二维数组的抽象，提供访问数组的语义时应该给出索引对应的实际对象，而不是对象的副本。

类 `GameBoard` 的实现定义了两个 `getPieceAt()` 函数，其中一个返回对 `GamePiece` 的引用，另一个返回对 `GamePiece` 的 `const` 引用。第 16 章将解释这种重载是如何做到的。

下面是类 `GameBoard` 的方法及静态成员的定义。其实现与第 9 章中类 `Spreadsheet` 的实现几乎完全相同。当然，成品代码应该在函数 `setPieceAt()` 和 `getPieceAt()` 中完成越界检查。在此省略了这部分代码，这不是本章要讨论的问题。

```

// GameBoard.cpp
#include "GameBoard.h"

const int GameBoard::kDefaultWidth;
const int GameBoard::kDefaultHeight;

GameBoard::GameBoard(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    mCells = new GamePiece* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new GamePiece[mHeight];
    }
}

GameBoard::GameBoard(const GameBoard& src)
{
    copyFrom(src);
}

GameBoard::~GameBoard()
{
    // Free the old memory
    for (int i = 0; i < mWidth; i++) {
        delete[] mCells[i];
    }

    delete[] mCells;
}

```



```

void GameBoard::copyFrom(const GameBoard& src)
{
    int i, j;
    mWidth = src.mWidth;
    mHeight = src.mHeight;

    mCells = new GamePiece* [mWidth];
    for (i = 0; i < mWidth; i++) {
        mCells[i] = new GamePiece[mHeight];
    }
    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}

GameBoard& GameBoard::operator=(const GameBoard& rhs)
{
    // Check for self-assignment
    if (this == &rhs) {
        return (*this);
    }
    // Free the old memory
    for (int i = 0; i < mWidth; i++) {
        delete[] mCells[i];
    }
    delete[] mCells;

    // Copy the new memory
    copyFrom(rhs);

    return (*this);
}

void GameBoard::setPieceAt(int x, int y, const GamePiece& inElem)
{
    mCells[x][y] = inElem;
}

GamePiece& GameBoard::getPieceAt(int x, int y)
{
    return (mCells[x][y]);
}

const GamePiece& GameBoard::getPieceAt(int x, int y) const
{
    return (mCells[x][y]);
}

```

类 GameBoard 可以很好地工作。如果你编写了类 ChessPiece，就可以创建 GameBoard 对象，并如下使用：

```

GameBoard chessBoard(10, 10);
ChessPiece pawn;

chessBoard.setPieceAt(0, 0, pawn);

```

### 模板 Grid 类

前一小节的类 GameBoard 尽管很不错，不过还是有些不足之处。比如，它和第 9 章的类 Spreadsheet 很像，但是如果要把 GameBoard 用作电子表格，惟一的办法就是让类 SpreadsheetCell 成为 GamePiece 的一个子类。然而这没有什么意义，因为它无法满足继承的 is-a 原则：SpreadsheetCell 不是 GamePiece。如果能够编写一个通用的 grid 类，能够用于 Spreadsheet 或 ChessBoard 等不同的目的，岂不是更好？在 C++ 中，编写类模板（template）可以实现这一点，采用类模板可以在不指定数据类型的情况下编写类。这样一来，客户代码可以指定自己需要使用的数据类型来实例化（instantiate）模板。

#### Grid 类的定义

要理解类模板，看一下它的语法会很有帮助。下面的例子就说明了如何修改类 GameBoard 来建立模板化的 Grid 类。不要对模板的语法感到害怕——在下面这段代码之后我们将一一解释。注意，类名已经由 GameBoard 变成了 Grid，方法 setPieceAt() 与 getPieceAt() 也变成了 setElementAt() 与 getElementAt()，以此来反映类 Grid 的更通用的本质。

```
// Grid.h
template <typename T>
class Grid
{
public:
    Grid(int inWidth = kDefaultWidth, int inHeight = kDefaultHeight);
    Grid(const Grid<T>& src);
    ~Grid();
    Grid<T>& operator=(const Grid<T>& rhs);

    void setElementAt(int x, int y, const T& inElem);
    T& getElementAt(int x, int y);
    const T& getElementAt(int x, int y) const;
    int getHeight() const { return mHeight; }
    int getWidth() const { return mWidth; }
    static const int kDefaultWidth = 10;
    static const int kDefaultHeight = 10;

protected:
    void copyFrom(const Grid<T>& src);
    T** mCells;
    int mWidth, mHeight;
};
```

上面已经给出了类的完整定义，再来一行一行地看一遍：

```
template <typename T>
```

第一行指出，下面的类定义是在一种类型上定义的模板。template 和 typename 都是 C++ 中的关键字。就像前面讨论的那样，类似于函数对值“参数化”，模板也采用同样的方法对数据类型进行了“参数化”。就像在函数中使用参数（parameter）名来代表调用该函数的程序所传递的实参（argument）一样，在模板中是使用类型名（比如 T）来代表调用该模板的程序所指定的数据类型。名字 T 并没有什么特别之处——只要愿意，可以使用任何名称。

出于历史原因，可以使用关键字 class 代替 typename 来指定模板的类型参数。因此，在许多书和代码中，使用了类似这样的语法：template <class T>。然而，在这里使用“class”会让人感到迷惑，

它暗示着数据类型必须是类，其实并不是这样。因此，在这本书中我们都使用 `typename`。

`template` 这个模板标识符要应用于整个语句，在这里就是要应用于类定义。

继续往下看几行代码，复制构造函数如下：

```
Grid (const Grid<T> & src);
```

可以看到，`src` 参数的类型不再是 `const Grid&`，而是 `const Grid<T>&`。在编写类模板时，原来认为是类名的 `Grid` 实际上是模板名。在讨论具体的 `Grid` 类或类型时，是把它们作为 `Grid` 类模板针对特定类型的实例化来讨论的，比如 `int`、`SpreadsheetCell` 或 `ChessPiece` 等。但是尚未指定具体类型，所以必须使用替代的模板参数 `T` 来表示以后可能会使用的任何类型。因此，需要指出 `Grid` 对象的一个类型作为传递给方法的参数或者作为方法的返回值时，就必须使用 `Grid<T>`。从赋值操作符、参数、返回值以及 `copyFrom()` 方法的参数都可以看到这种变化。

在类定义中，需要时编译器会把 `Grid` 解释为 `Grid<T>`。但是，最好还是养成明确指定 `Grid<T>` 的好习惯，因为在类外部引用由模板生成的类型时就会采用这种语法。

对这个类的最后一个修改是，`setElementAt()` 与 `getElementAt()` 等方法用类型 `T` 代替类型 `GamePiece` 作为参数和返回值的类型：

```
void setElementAt(int x, int y, const T& inElem);
T& getElementAt(int x, int y);
const T& getElementAt(int x, int y) const;
```

这个类型 `T` 是用户指定的任意数据类型的占位符。`mCells` 的类型现在是类型 `T**` 而不是类型 `GameBoard**` 了，因为不管用户指定的类型 `T` 是什么，`mCells` 都将指向动态分配的 `T` 类型的二维数组。

模板类可以包含内联方法，比如 `getHeight()` 和 `getWidth()`。

#### Grid 类的方法定义

对于 `Grid` 模板，限定符 `template <typename T>` 必须写在所有方法的定义之前。构造函数如下：

```
template <typename T>
Grid<T>::Grid(int inWidth, int inHeight) : mWidth(inWidth), mHeight(inHeight)
{
    mCells = new T* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new T[mHeight];
    }
}
```

注意，`::` 之前的类名为 `Grid<T>`，而不是 `Grid`。在你自己定义的所有方法和静态数据成员定义中，必须指定 `Grid<T>` 作为类名。除了用占位符类型 `T` 代替 `GamePiece` 类型之外，构造函数体与 `GameBoard` 的构造函数是完全相同的。

其他的方法和静态数据成员定义也与类 `GameBoard` 中相应的部分类似，只是 `template` 和 `Grid<T>` 语法上有所改变：

```
template <typename T>
const int Grid<T>::kDefaultWidth;

template <typename T>
const int Grid<T>::kDefaultHeight;
```

```
template <typename T>
Grid<T>::Grid(const Grid<T>& src)
{
    copyFrom(src);
}
```

```
template <typename T>
Grid<T>::~Grid()
{
    // Free the old memory.
    for (int i = 0; i < mWidth; i++) {
        delete [] mCells[i];
    }
    delete [] mCells;
}
```

```
template <typename T>
void Grid<T>::copyFrom(const Grid<T>& src)
{
```

```
    int i, j;
    mWidth = src.mWidth;
    mHeight = src.mHeight;
```

```
    mCells = new T* [mWidth];
    for (i = 0; i < mWidth; i++) {
        mCells[i] = new T[mHeight];
    }
```

```
    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
```

```
}
```

```
template <typename T>
Grid<T>& Grid<T>::operator=(const Grid<T>& rhs)
```

```
{
    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }
    // Free the old memory.
    for (int i = 0; i < mWidth; i++) {
        delete [] mCells[i];
    }
    delete [] mCells;
    // Copy the new memory.
    copyFrom(rhs);

    return (*this);
}
```

```
template <typename T>
void Grid<T>::setElementAt(int x, int y, const T& inElem)
{
```

```

        mCells[x][y] = inElem;
    }

```

```

template <typename T>
T& Grid<T>::getElementAt(int x, int y)
{
    return (mCells[x][y]);
}

```

```

template <typename T>
const T& Grid<T>::getElementAt(int x, int y) const
{
    return (mCells[x][y]);
}

```

### 使用 Grid 模板

要创建网格对象时，不能单独使用 Grid 作为数据类型，必须指定存储在该 Grid 对象中的数据类型。针对具体数据类型创建模板类的一个对象，称为对模板实例化。下面是一个例子：

```

#include "Grid.h"

int main(int argc, char** argv)
{
    Grid<int> myIntGrid; // Declares a grid that stores ints
    myIntGrid.setElementAt(0, 0, 10);
    int x = myIntGrid.getElementAt(0, 0);

    Grid<int> grid2(myIntGrid);
    Grid<int> anotherIntGrid = grid2;

    return (0);
}

```

需要注意的是，类型 myIntGrid、grid2 和 anotherIntGrid 类型都是 Grid<int>。在这些网格中不能存储 SpreadsheetCell 或 ChessPiece。如果非要这样做，编译器就会报错。

变量的类型声明是很重要的，下面这两行代码都不能编译通过：

```

Grid test; // WILL NOT COMPILE
Grid<> test; // WILL NOT COMPILE

```

第一行代码会导致编译器报错：“使用类模板需要提供模板参数列表”。而第二行代码会导致编译器指出：“模板参数的数目不对”。

如果要声明取 Grid 对象为参数的函数或方法，必须指定存储在该网格中的数据类型作为 Grid 类型的一部分：

```

void processIntGrid(Grid<int>& inGrid)
{
    // Body omitted for brevity
}

```

Grid 模板不仅仅能够存储 int 数据类型。例如，还可以实例化 Grid 来存储 SpreadsheetCell。

```

Grid<SpreadsheetCell> mySpreadsheet;
SpreadsheetCell myCell;

```



```
mySpreadsheet.setElementAt(3, 4, myCell);
```

它也可以存储指针类型:

```
Grid<char*> myStringGrid;  
myStringGrid.setElementAt(2, 2, "hello");
```

指定的类型甚至可以是另一个模板类型。下面这个例子就使用了标准模板库中的向量模板 (在第 4 章介绍过):

```
Grid<vector<int> > gridOfVectors; // Note the extra space!  
vector<int> myVector;  
gridOfVectors.setElementAt(5, 6, myVector);
```

在模板嵌套中, 必须在两个相邻的尖括号之间留出一个空格。在 C++ 中要求使用这样的语法, 主要是因为, 在下面这样的例子中, 编译器会把 >> 解释为 I/O 流析取操作符。

```
Grid<vector<int>>> gridOfVectors; // INCORRECT SYNTAX
```

还可以在栈上动态分配 Grid 模板的实例化:

```
Grid<int>* myGridp = new Grid<int>();  
myGridp->setElementAt(0, 0, 10);  
int x = myGridp->getElementAt(0, 0);  
  
delete myGridp;
```

### 11.2.2 编译器如何处理模板

为了了解模板的复杂性, 需要学习一下编译器如何处理模板代码。编译器遇到模板方法定义时, 它首先进行语法检查, 但是并不实际编译模板。编译器不能编译模板定义, 因为它并不知道模板将用于什么数据类型。让编译器在不知道  $x$  和  $y$  数据类型的情况下为  $x = y$  生成代码是不可能的。

编译器遇到模板的实例化时, 比如 `Grid<int> myIntGrid`, 会使用 `int` 代替模板类定义中的所有 `T`, 来编写 Grid 模板的 `int` 版代码。编译器遇到模板的不同实例化时, 比如 `Grid<SpreadsheetCell> mySpreadsheet`, 则为 `SpreadsheetCell` 编写另一个 Grid 类。如果编程语言不支持模板, 就要求必须为每种元素类型编写独立的类, 此时编译器就只是编译你编写的代码。所以模板并没有什么不可思议的地方, 它只是自动处理了这些繁琐的过程。如果在程序中没有针对任何类型对类模板实例化, 则永远也不会编译类方法定义。

这个实例化过程解释了为什么需要在类定义的所有地方都要使用 `Grid<T>` 语法。当编译器为特定类型 (比如 `int`) 实例化模板时, 它会使用 `int` 代替 `T`, 这样类型就变成了 `Grid<int>`。

#### 选择性实例化

在为多种不同的数据类型实例化模板时, 由于编译器会给每种数据类型都生成一个模板代码的副本, 所以为多种不同数据类型实例化模板可能会导致代码膨胀。使用模板时, 最后很可能得到极其庞大的可执行文件。

不过编译器只会为针对特定类型实际调用的类方法生成代码, 所以这个问题有所缓解。例如, 对于上面给出的 Grid 模板类, 假设在 `main()` 中编写了下面这段代码 (而且只编写了这段代码),

```
Grid<int> myIntGrid;  
myIntGrid.setElementAt(0, 0, 10);
```



那么, 编译器只会为 Grid 的 int 版本产生无参数构造函数、析构函数和 setElementAt() 方法, 而不会生成其他的方法, 比如复制构造函数、赋值操作符或 getHeight() 等。

#### 类型方面的模板要求

在编写独立于类型的代码时, 必须对这些类型做一定的假设。比如, 在 Grid 模板中, 由于存在这么一行代码: mCells[x][y] = inElem, 所以要假设元素类型 (用 T 表示) 有赋值操作符。同理, 还可以假设它有默认构造函数, 从而允许创建一个元素数组。

如果要使用某种数据类型对模板实例化, 而对于特定程序中的模板来说, 该数据类型并不支持模板中的全部操作, 则代码不能编译成功。然而, 即使要使用的数据类型不支持模板代码所需的全部操作, 也可以采用选择性实例化来使用其中的一部分方法。比如, 要为没有赋值操作符的对象创建网格, 但是永远不会在这个网格上调用 setElementAt(), 那么代码也可以正常运行。但是只要试图调用 setElementAt(), 就会产生编译错误。

### 11.2.3 模板代码在文件之间的分布

通常都是把类定义放在头文件中, 方法的定义放在源文件中。创建和使用类对象的代码使用 “#include” 包含头文件, 并通过连接程序来访问方法代码。但是模板不是这样工作的。模板是编译器用于为实例化数据类型生成实际方法的 “模板”, 所以在模板类定义和方法定义的任何源文件中, 模板类定义和方法定义必须对编译器都是可用的。从这个意义上讲, 模板类的方法有些类似于内联方法。可以采用几种机制来实现这种包含。

#### 在头文件中定义模板

类方法的定义可以直接放在定义类的同一个头文件中。在使用模板的源文件中用 “#include” 包含这个文件时, 编译器会访问它所需要的全部代码。

另一种方法是, 可以把模板方法定义放在一个单独的头文件中, 并在类定义所在的头文件中用 “#include” 包含这个头文件。但是要确保包含方法定义的 #include 语句放在类定义的后面, 否则代码不能编译通过。

```
// Grid.h

template <typename T>
class Grid
{
    // Class definition omitted for brevity
};

#include "GridDefinitions.h"
```

这样有助于保持类定义与方法定义相分离。

#### 在源文件中定义模板

在头文件中放方法实现看起来有点奇怪。如果不喜欢这种语法, 还有一种方法, 可以把方法的定义放在源文件中。但是仍然需要保证该方法定义对于使用模板的代码是可用的, 为此, 可以在模板类定义头文件中用 “#include” 包含方法实现源 (source) 文件。在亲眼目睹之前, 乍听起来这有点奇怪, 但是在 C++ 中确实是合法的。头文件如下:

```
// Grid.h

template <typename T>
```

```
class Grid
{
    // Class definition omitted for brevity
};

#include "Grid.cpp"
```

C++ 标准实际定义了一种方法，可以把模板方法定义放在源文件中，而不需要通过“#include”包含在头文件中。这要使用关键字 `export` 来指定，模板定义应该在所有的翻译单元（translation unit）（源文件）中都可用。遗憾的是，在写这本书的时候，还没有几个商用编译器支持这个特性，看起来许多开发商近期好像都没有要支持它的打算。

#### 11.2.4 模板参数

在类 `Grid` 的例子中，模板 `Grid` 只有一个模板参数（template parameter）：即存储在网格中的数据类型的。在编写类模板时，要用尖括号指定参数列表，就像这样：

```
template <typename T>
```

这个参数列表类似于函数或方法中的参数列表。就像在函数和方法中一样，只要愿意，可以编写带有多个模板参数的类。另外，这些参数不必都是类型参数，而且它们可以有默认值。

##### 无类型模板参数

无类型参数是诸如 `int` 和指针这样的“常规”参数，也就是你熟悉的函数和方法中的参数。不过，模板允许无类型参数取“简单”类型的值：`int`、`enum`、指针和引用。

在模板类 `Grid` 中，可以使用无类型模板参数指定网格的高度和宽度，而不是在构造函数中指定这些参数。与在构造函数中指定无类型参数相比，在模板列表指定无类型参数的主要优点是，代码在编译之前就已经知道参数的值了。回忆一下，编译器是通过在编译前替换模板参数来为模板化方法生成代码的。因此，在实现中可以使用常规的二维数组，而不是动态地分配数组。下面是新的类定义：

```
template <typename T, int WIDTH, int HEIGHT>
class Grid
{
public:
    void setElementAt(int x, int y, const T& inElem);
    T& getElementAt(int x, int y);
    const T& getElementAt(int x, int y) const;
    int getHeight() const { return HEIGHT; }
    int getWidth() const { return WIDTH; }

protected:
    T mCells[WIDTH][HEIGHT];
};
```

这个类比原来的类要简单得多。注意，模板参数列表需要三个参数：存储在网格中的对象的类型和网格的高度与宽度。宽度和高度用来创建一个二维数组来存储这些对象。在这个类中没有动态分配的内存，所以也就不需要用户定义的复制构造函数、析构函数或赋值操作符了。实际上，甚至不需要编写默认的构造函数，编译器生成的构造函数就足够了。下面是类方法的定义：

```
template <typename T, int WIDTH, int HEIGHT>
void Grid<T, WIDTH, HEIGHT>::setElementAt(int x, int y, const T& inElem)
```

```
{
    mCells[x][y] = inElem;
}
```

```
template <typename T, int WIDTH, int HEIGHT>
T& Grid<T, WIDTH, HEIGHT>::getElementAt(int x, int y)
{
    return (mCells[x][y]);
}
```

```
template <typename T, int WIDTH, int HEIGHT>
const T& Grid<T, WIDTH, HEIGHT>::getElementAt(int x, int y) const
{
    return (mCells[x][y]);
}
```

注意，在前面指定 `Grid<T>` 的地方，现在都必须指定 `Grid<T, WIDTH, HEIGHT>` 来表示这三个模板参数。

可以实例化这个模板，并如下使用：

```
Grid<int, 10, 10> myGrid;
Grid<int, 10, 10> anotherGrid;

myGrid.setElementAt(2, 3, 45);
anotherGrid = myGrid;

cout << anotherGrid.getElementAt(2, 3);
```

这段代码看起来非常漂亮。虽然在声明 `Grid` 时语法上稍微有点凌乱，但是具体的 `Grid` 代码十分简洁。遗憾的是，出乎原先的预想，在此存在很多限制。首先，不能用非常量整数来指定网格的高度或宽度。下面这段代码不能编译通过：

```
int height = 10;
Grid<int, 10, height> testGrid; // DOES NOT COMPILE
```

但是如果把高度设置为 `const`，则可以编译通过：

```
const int height = 10;
Grid<int, 10, height> testGrid; // compiles and works
```

第二个问题更重要。既然网格高度和宽度都是模板参数，它们就是每个网格类型的一部分。这意味着 `Grid<int, 10, 10>` 与 `Grid<int, 10, 11>` 是两个不同的类型。不能把一种类型的对象赋值给另一种类型的对象，一种类型的变量也不能传递给希望使用另一种类型变量的函数或方法。

无类型模板参数变成了实例化对象类型规范的一部分。

### 整数无类型参数的默认值

如果继续采用这种方法，把网格的高度和宽度作为模板参数，可能需要像前面在类 `Grid<T>` 的构造函数中所做的那样，为高度和宽度提供默认值。C++ 允许使用类似的语法给模板参数提供默认值。下面是类定义：

```
template <typename T, int WIDTH = 10, int HEIGHT = 10>
class Grid
{
    // Remainder of the implementation is identical to the previous version
};
```

但是并不需要在方法定义的模板规范中为 WIDTH 和 HEIGHT 指定默认值。例如，下面是 setElementAt() 的实现：

```
template <typename T, int WIDTH, int HEIGHT>
void Grid<T, WIDTH, HEIGHT>::setElementAt(int x, int y, const T& inElem)
{
    mCells[x][y] = inElem;
}
```

现在可以仅仅使用元素类型来实例化 Grid，也可以使用元素类型和宽度来实例化 Grid，或者使用元素类型、宽度和高度来实例化 Grid：

```
Grid<int> myGrid;
Grid<int, 10> anotherGrid;
Grid<int, 10, 10> aThirdGrid;
```

对于模板参数列表中的默认参数，其规则与函数或方法的默认参数规则相同，可以按照从右开始的顺序提供参数的默认值。

### 11.2.5 方法模板

C++ 允许对类的单个方法进行模板化。这些方法可以放在类模板中，也可以放在非模板化类中。在编写模板化的类方法时，实际上是针对多种不同的数据类型编写该方法的多个不同版本。方法模板对于类模板中的赋值操作符和复制构造函数用处较大。

不能模板化虚方法和析构函数。

看一下只有一个参数的原 Grid 模板：即仅有一个元素类型参数。你可以实例化多种不同类型的网格，比如 int 和 double：

```
Grid<int> myIntGrid;
Grid<double> myDoubleGrid;
```

但是，Grid<int> 和 Grid<double> 是两种不同的类型。如果编写了一个函数，它取 Grid<double> 类型的对象作为参数，就不能为这个函数传递 Grid<int> 对象。即使知道 int 网格可以复制到 double 网格也不行，因为 int 不能强制转换为 double，也不能把 Grid<int> 类型的对象赋值给 Grid<double> 类型的对象，或者从 Grid<int> 构造一个 Grid<double>。下面这两行代码都不能编译通过：

```
myDoubleGrid = myIntGrid; // DOES NOT COMPILE
Grid<double> newDoubleGrid(myIntGrid); // DOES NOT COMPILE
```

问题在于，Grid 模板的复制构造函数和 operator= 签名如下：

```
Grid(const Grid<T>& src);
Grid<T>& operator=(const Grid<T>& rhs);
```

Grid 复制构造函数和 operator= 都采用了 const Grid<T> 的引用。在实例化一个 Grid<double> 并企图调用其复制构造函数和 operator= 时, 编译器会使用这些签名生成如下方法:

```
Grid(const Grid<double>& src);
Grid<double>& operator=(const Grid<double>& rhs);
```

注意, 在生成的 Grid<double> 类中, 并没有采用 Grid<int> 作为参数的构造函数或 operator=。但是, 可以为类 Grid 增加复制构造函数和 operator= 的模板化版本来生成一些例程, 从而可以由一种网格类型转换为另一种类型的网格, 以此弥补这个疏忽。下面是类 Grid 的新定义:

```
template <typename T>
class Grid
{
public:
    Grid(int inWidth = kDefaultWidth, int inHeight = kDefaultHeight);
    Grid(const Grid<T>& src);
    template <typename E>
    Grid(const Grid<E>& src);
    ~Grid();

    Grid<T>& operator=(const Grid<T>& rhs);
    template <typename E>
    Grid<T>& operator=(const Grid<E>& rhs);

    void setElementAt(int x, int y, const T& inElem);
    T& getElementAt(int x, int y);
    const T& getElementAt(int x, int y) const;

    int getHeight() const { return mHeight; }
    int getWidth() const { return mWidth; }

    static const int kDefaultWidth = 10;
    static const int kDefaultHeight = 10;

protected:
    void copyFrom(const Grid<T>& src);
    template <typename E>
    void copyFrom(const Grid<E>& src);

    T** mCells;
    int mWidth, mHeight;
};
```

成员模板不会取代同名的非模板成员。由于编译器生成的代码可能有不同版本, 这个规则导致了复制构造函数和 operator= 会存在问题。如果编写了复制构造函数和 operator= 的模板化版本, 并去掉了非模板化的构造函数和 operator=, 编译器不会为同一类型的网格赋值运算调用这些新的模板化构造函数和 operator=。相反, 编译器会生成一个复制构造函数和 operator=, 来完成两个同类型网格的创建和赋值, 而这并不是你想要的。因此, 还必须保留老的非模板化复制构造函数和 operator=。

首先看一下新的模板化复制构造函数的签名:

```
template <typename E>
Grid(const Grid<E>& src);
```



可以看到，这里用不同的类型名 E (“element” 的简写) 声明了另一个模板。这个类针对一种类型 T 模板化，另外新的复制构造函数还针对另一种不同的类型 E 模板化。基于这种双重模板化，可以把一种类型的网格复制为另一种类型。

下面是新复制构造函数的定义：

```
template <typename T>
template <typename E>
Grid<T>::Grid(const Grid<E>& src)
{
    copyFrom(src);
}
```

可以看到，在声明成员模板的代码行（使用参数 E）之前必须有声明类模板的代码行（使用参数 T）。不能把它们像下面这样组合：

```
template <typename T, typename E> // INCORRECT TEMPLATE PARAMETER LIST!
Grid<T>::Grid(const Grid<E>& src)
```

一些编译器要求在类定义中内联提供方法模板定义，但是 C++ 标准允许在类定义外部定义方法模板。

复制构造函数使用保护的（protected）copyFrom() 方法，所以该类还需要一个模板化的 copyFrom() 方法：

```
template <typename T>
template <typename E>
void Grid<T>::copyFrom(const Grid<E>& src)
{
    int i, j;
    mWidth = src.getWidth();
    mHeight = src.getHeight();

    mCells = new T* [mWidth];
    for (i = 0; i < mWidth; i++) {
        mCells[i] = new T[mHeight];
    }
    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = src.getElementAt(i, j);
        }
    }
}
```

除了 copyFrom() 方法定义之前额外的模板参数行外，需要注意的一点是，必须使用公共的存取方法 getWidth()、getHeight() 和 getElementAt() 来访问 src 的元素。这是因为正在复制的对象是 Grid<T> 类型的对象，而源对象是 Grid<E> 类型的对象。它们的类型不同，所以必须求助于公共方法。

最后一个模板化方法是赋值操作符。注意，它取了一个 const Grid<E>&，但是返回的却是 Grid<T>&。

```
template <typename T>
template <typename E>
Grid<T>& Grid<T>::operator=(const Grid<E>& rhs)
```



```

{
    // Free the old memory.
    for (int i = 0; i < mWidth; i++) {
        delete [] mCells[i];
    }
    delete [] mCells;

    // Copy the new memory.
    copyFrom(rhs);

    return (*this);
}

```

不需要在模板化赋值操作符中检查自赋值，因为同一类型的赋值还是在原来的非模板化 `operator=` 中发生，所以在这里不会碰到自赋值的情况。

方法模板除了令人迷惑的语法之外，还有另外一个问题，一些编译器没有实现对方模板的完全（或任何）支持。在你选择的编译器上试一下这个例子，看能否使用这个特性。

#### 带有无类型参数的方法模板

在前面使用 `HEIGHT` 和 `WIDTH` 整数模板参数的例子中，我们提到，主要的问题是高度和宽度会变成类型的一部分。由于存在这个限制，所以不能把有一种高度和宽度的网格赋值给另一个高度和宽度不同的网格。但是，在某些情况下，需要把一个大小的网格赋值或者复制给另一个大小的网格。可能需要只复制源数组中适合目标数组的元素，而不是让目标对象成为源对象的完全克隆，如果源数组的维数小于目标数组的维数，可以使用默认值填充目标数组。有了赋值操作符和复制构造函数的方法模板，就可以正确做到这一点，这样就能完成不同大小网格之间的赋值和复制了。

下面是类定义：

```

template <typename T, int WIDTH = 10, int HEIGHT = 10>
class Grid
{
public:
    Grid() {}

    template <typename E, int WIDTH2, int HEIGHT2>
    Grid(const Grid<E, WIDTH2, HEIGHT2>& src);

    template <typename E, int WIDTH2, int HEIGHT2>
    Grid<T, WIDTH, HEIGHT>& operator=(const Grid<E, WIDTH2, HEIGHT2>& rhs);

    void setElementAt(int x, int y, const T& inElem);
    T& getElementAt(int x, int y);
    const T& getElementAt(int x, int y) const;
    int getHeight() const { return HEIGHT; }
    int getWidth() const { return WIDTH; }

protected:
    template <typename E, int WIDTH2, int HEIGHT2>
    void copyFrom(const Grid<E, WIDTH2, HEIGHT2>& src);

    T mCells[WIDTH][HEIGHT];
};

```

我们已经为复制构造函数和赋值操作符增加了方法模板，还提供了一个辅助方法 `copyFrom()`。回忆一下第 8 章，如果已经编写了复制构造函数，编译器就不再生成默认的构造函数，所以还必须增加默认

构造函数。但是要注意，我们并不需要编写非模板化的复制构造函数和赋值操作符方法，因为编译器还会生成这些方法。它们只是简单地把 mCells 从源复制到或者赋值到目标，如果要完成两个相同大小网格的赋值或复制，编译器生成的复制构造函数和赋值操作符正好提供了我们所需要的语义。

在对复制构造函数、赋值操作符和 copyFrom() 方法进行模板化时，必须指定所有的三个模板参数。模板化的复制构造函数如下：

```
template <typename T, int WIDTH, int HEIGHT>
template <typename E, int WIDTH2, int HEIGHT2>
Grid<T, WIDTH, HEIGHT>::Grid(const Grid<E, WIDTH2, HEIGHT2>& src)
{
    copyFrom(src);
}
```

以下是 copyFrom() 和 operator= 的实现。注意，copyFrom() 只是从 src 分别复制 x 和 y 两维中的 WIDTH 和 HEIGHT 元素，即使 src 比目标网格大也是如此。如果在任意一维上 src 较小，copyFrom() 则用 0 初始值来填充多出来的点。如果 T 是一个类的类型，T() 就会为该对象调用默认的构造函数，如果 T 是简单类型，则生成 0。这个语法叫做零初始化 (zero-initialization) 语法。要给类型未知的变量提供合理的默认值，这是一个很好的做法。

```
template <typename T, int WIDTH, int HEIGHT>
template <typename E, int WIDTH2, int HEIGHT2>
void Grid<T, WIDTH, HEIGHT>::copyFrom(const Grid<E, WIDTH2, HEIGHT2>& src)
{
    int i, j;
    for (i = 0; i < WIDTH; i++) {
        for (j = 0; j < HEIGHT; j++) {
            if (i < WIDTH2 && j < HEIGHT2) {
                mCells[i][j] = src.getElementAt(i, j);
            } else {
                mCells[i][j] = T();
            }
        }
    }
}

template <typename T, int WIDTH, int HEIGHT>
template <typename E, int WIDTH2, int HEIGHT2>
Grid<T, WIDTH, HEIGHT>& Grid<T, WIDTH, HEIGHT>::operator=(
    const Grid<E, WIDTH2, HEIGHT2>& rhs)
{
    // No need to check for self-assignment because this version of
    // assignment is never called when T and E are the same

    // No need to free any memory first

    // Copy the new memory.
    copyFrom(rhs);
    return (*this);
}
```

### 11.2.6 模板类特殊化

可以为特定类型提供其他的类模板实现。例如，你可能确定针对 char\* (C 风格的字符串) 的 Grid

行为是无意义的。网格当前存储的是指针类型的浅副本。对于 `char*`，字符串的深复制可能才有意义。

模板的其他实现称为模板特殊化 (template specialization)。这里的语法看起来也有点古怪。在特殊化模板类时，必须指定它是模板，而且要指定你正在为这个特定类型编写模板。下面是针对 `char*` 对原 `Grid` 模板类完成特殊化的语法。

```
// #includes for working with the C-style strings.
#include <cstdlib>
#include <cstring>
using namespace std;

// When the template specialization is used, the original template must be visible
// too. #including it here ensures that it will always be visible when this
// specialization is visible.
#include "Grid.h"

template <>
class Grid<char*>
{
public:
    Grid(int inWidth = kDefaultWidth, int inHeight = kDefaultHeight);
    Grid(const Grid<char*>& src);
    ~Grid();
    Grid<char*>& operator=(const Grid<char*>& rhs);

    void setElementAt(int x, int y, const char* inElem);
    char* getElementAt(int x, int y) const;

    int getHeight() const { return mHeight; }
    int getWidth() const { return mWidth; }
    static const int kDefaultWidth = 10;
    static const int kDefaultHeight = 10;
protected:
    void copyFrom(const Grid<char*>& src);

    char*** mCells;
    int mWidth, mHeight;
};
```

注意，在特殊化中没有引用任何类型变量，比如 `T`，而是直接使用 `char*`。在这一点上，你可能会问这样一个明显的问题：为什么这个类仍然是模板。也就是说，语法设置成这样有什么好处？

```
template <>
class Grid<char*>
```

通过这个语法，告诉编译器这个类是类 `Grid` 的 `char*` 特殊化。假设没有使用这个语法，而仅仅写作为：

```
class Grid
```

编译器是不会让你这样做的，因为这里已经有一个名为 `Grid` 的类（原来的模板类）。只有通过特殊化才能重用这个名字。特殊化的主要好处是对于用户来说它们是不可见的。当用户创建 `int` 或 `Spreadsheet-Cell` 的 `Grid` 时，编译器会从原来的 `Grid` 模板生成代码。当用户创建 `char*` 的 `Grid` 时，编译器要使用 `char*` 特殊化。这些全部都在后台进行。

```

Grid<int> myIntGrid; // Uses original Grid template
Grid<char*> stringGrid1(2, 2); // Uses char* specialization

char* dummy = new char[10];

strcpy(dummy, "dummy");

stringGrid1.setElementAt(0, 0, "hello");
stringGrid1.setElementAt(0, 1, dummy);
stringGrid1.setElementAt(1, 0, dummy);
stringGrid1.setElementAt(1, 1, "there");

delete[] dummy;

Grid<char*> stringGrid2(stringGrid1);

```

在特殊化模板时，你没有“继承”任何代码：特殊化不像子类，必须重写类的整个实现。这里不要提供同名或有相同行为的方法。实际上，你也可以编写一个完全不同的类，它可以与原来的类没有任何关系。当然，这就会滥用模板的特殊化能力，不管出于什么原因都不能这样做。下面是 `char*` 特殊化的方法实现。不同于原来的模板定义，在此每个方法或静态方法定义之前不用重复写上 `template<>`。

```

const int Grid<char*>::kDefaultWidth;
const int Grid<char*>::kDefaultHeight;

Grid<char*>::Grid(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    mCells = new char** [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new char* [mHeight];
        for (int j = 0; j < mHeight; j++) {
            mCells[i][j] = NULL;
        }
    }
}

```

```

Grid<char*>::Grid(const Grid<char*>& src)
{
    copyFrom(src);
}

```

```

Grid<char*>::~~Grid()
{
    // Free the old memory.
    for (int i = 0; i < mWidth; i++) {
        for (int j = 0; j < mHeight; j++) {
            delete[] mCells[i][j];
        }
        delete[] mCells[i];
    }
    delete[] mCells;
}

```

```

void Grid<char*>::copyFrom(const Grid<char*>& src)
{

```

```

int i, j;
mWidth = src.mWidth;
mHeight = src.mHeight;
mCells = new char** [mWidth];
for (i = 0; i < mWidth; i++) {
    mCells[i] = new char* [mHeight];
}

for (i = 0; i < mWidth; i++) {
    for (j = 0; j < mHeight; j++) {
        if (src.mCells[i][j] == NULL) {
            mCells[i][j] = NULL;
        } else {
            mCells[i][j] = new char[strlen(src.mCells[i][j]) + 1];
            strcpy(mCells[i][j], src.mCells[i][j]);
        }
    }
}

```

```

Grid<char*>& Grid<char*>::operator=(const Grid<char*>& rhs)

```

```

{
    int i, j;
    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }
    // Free the old memory.
    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            delete[] mCells[i][j];
        }
        delete[] mCells[i];
    }
    delete[] mCells;

    // Copy the new memory.
    copyFrom(rhs);

    return (*this);
}

```

```

void Grid<char*>::setElementAt(int x, int y, const char* inElem)

```

```

{
    delete[] mCells[x][y];
    if (inElem == NULL) {
        mCells[x][y] = NULL;
    } else {
        mCells[x][y] = new char[strlen(inElem) + 1];
        strcpy(mCells[x][y], inElem);
    }
}

```

```

char* Grid<char*>::getElementAt(int x, int y) const

```

```

{
    if (mCells[x][y] == NULL) {
        return (NULL);
    }
}

```



```

    }
    char* ret = new char[strlen(mCells[x][y]) + 1];
    strcpy(ret, mCells[x][y]);

    return (ret);
}

```

方法 getElementAt() 返回该字符串的深复制，所以不需要返回 const char\* 的重载方法。

### 11.2.7 从模板类派生子类

可以编写模板类的子类。如果子类从模板本身继承，它必须也是模板。或者，可以编写从模板类特定实例化继承的子类，在这种情况下，不要求子类是模板。像前面的例子一样，假设你确定通用 Grid 类不能提供足够的功能作为游戏棋盘使用。而且想给游戏棋盘增加一个方法 move()，用来在棋盘上移动棋子。下面是 GameBoard 模板的类定义：

```

#include "Grid.h"

template <typename T>
class GameBoard : public Grid<T>
{
public:
    GameBoard(int inWidth = Grid<T>::kDefaultWidth,
              int inHeight = Grid<T>::kDefaultHeight);
    void move(int xSrc, int ySrc, int xDest, int yDest);
};

```

这个 GameBoard 模板是 Grid 模板的子类，因此继承了 Grid 的所有功能。不需要重写 setElementAt()、getElementAt() 或者其他的任何方法。也不需要增加复制构造函数、operator= 或者析构函数，这是因为在 GameBoard 中没有任何动态分配的内存。Grid 超类的复制构造函数、operator= 和析构函数会负责管理超类 Grid 中动态分配的内存。

除了超类是 Grid<T> 不是 Grid 之外，这里的继承语法看起来没有什么异常。之所以要使用这样的语法，原因在于，GameBoard 模板实际上并不是从通用 Grid 模板派生的子类。更合适的说法应该是，GameBoard 模板针对特定类型的各个实例化是派生自 Grid 模板针对该类型实例化的子类。例如，如果要使用类型 ChessPiece 实例化 GameBoard，编译器也会为 Grid<ChessPiece> 生成代码。语法：“public Grid<T>” 是说，这个类是从针对 T 类型参数的任何 Grid 实例化派生得来的子类。注意，C++ 对于模板继承有一定的命名查找规则，这些规则要求你必须指定：已经在超类 Grid<T> 中声明了 kDefaultWidth 和 kDefaultHeight，相应地 kDefaultWidth 和 kDefaultHeight 要依赖于 Grid<T>。

以下是构造函数和 move 方法的实现。还要注意超类构造函数调用中使用了 Grid<T>。此外，虽然许多编译器不能强制实现这一点，但是命名查找规则要求，要使用 this 指针来引用这个超类中的数据成员和方法。

```

template <typename T>
GameBoard<T>::GameBoard(int inWidth, int inHeight) :
    Grid<T>(inWidth, inHeight)
{
}

template <typename T>
void GameBoard<T>::move(int xSrc, int ySrc, int xDest, int yDest)

```



```
{
    this->mCells[xDest][yDest] = this->mCells[xSrc][ySrc];
    this->mCells[xSrc][ySrc] = T(); // zero-initialize the src cell
}
```

可以看到，`move()` 使用了“带有无类型参数的方法模板”小节中描述的 0 初始化语法 `T()`。可以如下使用 `GameBoard` 模板：

```
GameBoard<ChessPiece> chessBoard;

ChessPiece pawn;
chessBoard.setElementAt(0, 0, pawn);
chessBoard.move(0, 0, 0, 1);
```

### 11.2.8 继承与特殊化的区别

有些程序员发现，很难区分模板继承和模板特殊化。表 11-1 总结了模板继承与模板特殊化者之间的区别。

表 11-1

	继 承	特 殊 化
重用代码	是：子类包含了超类的所有数据成员和方法	否：在特殊化中必须重写所有的代码
重用名字	否：子类名必须不同于超类名	是：特殊化必须使用与原模板相同的名字
支持多态性	是：子类的对象可以代替超类的对象	否：模板针对一种类型的每个实例化都是一个不同的类型

要采用继承来扩展实现以及实现多态性，而使用特殊化为特定类型建立定制实现。

## 11.3 函数模板

除了编写类模板和类方法模板之外，也可以为独立的函数编写模板。例如，可以编写一个通用函数在数组中查找一个值，并返回其索引。

```
template <typename T>
int Find(T& value, T* arr, int size)
{
    for (int i = 0; i < size; i++) {
        if (arr[i] == value) {
            // Found it; return the index
            return (i);
        }
    }
    // Failed to find it; return -1
    return (-1);
}
```

`Find()` 函数模板可以对任意类型的数组进行操作。例如，可以用来在 `int` 数组中寻找一个 `int` 元素的索引，也可以在 `SpreadsheetCell` 数组中寻找一个 `SpreadsheetCell` 元素的索引。

调用该函数有两种方法：用尖括号明确指定类型；或者省略类型，让编译器自己根据实参来推导

(deduce) 参数是什么类型。下面是几个例子。

```
int x = 3, intArr[4] = {1, 2, 3, 4};
double d1 = 5.6, dArr[4] = {1.2, 3.4, 5.7, 7.5};

int res;
res = Find(x, intArr, 4); // Calls Find<int> by deduction
res = Find<int>(x, intArr, 4); // call Find<int> explicitly.

res = Find(d1, dArr, 4); // Call Find<double> by deduction.
res = Find<double>(d1, dArr, 4); // Calls Find<double> explicitly.

res = Find(x, dArr, 4); // DOES NOT COMPILE! Arguments are different types.

SpreadsheetCell c1(10), c2[2] = {SpreadsheetCell(4), SpreadsheetCell(10)};

res = Find(c1, c2, 2); // calls Find<SpreadsheetCell> by deduction
res = Find<SpreadsheetCell>(c1, c2, 2); // Calls Find<SpreadsheetCell>
// explicitly.
```

像类模板一样，函数模板也可以使用无类型参数。为简洁起见，在此仅仅给出函数模板的一个类型参数的例子。

C++标准库提供了一个模板化的 find() 函数，它的功能比上面这个函数强大的多。详细的内容请阅读第 22 章。

### 11.3.1 函数模板特殊化

就像特殊化类模板一样，也可以特殊化函数模板。例如，你可能需要给 C 风格的 char\* 字符串编写一个函数 Find()，它使用 strcmp() 代替 operator== 来比较字符串。下面就是用函数 Find() 的特殊化来实现这个功能。

```
template<>
int Find<char*>(char*& value, char** arr, int size)
{
    for (int i = 0; i < size; i++) {
        if (strcmp(arr[i], value) == 0) {
            // Found it; return the index
            return (i);
        }
    }
    // Failed to find it; return -1
    return (-1);
}
```

如果可以从实参推导出参数类型，可以省略函数名中的 <char\*>，这样可以得到以下原型：

```
template<>
int Find(char*& value, char** arr, int size)
```

然而，当涉及到重载方法时，推导规则相当错综复杂。所以，为了避免错误，最好明确指出类型。

虽然这个特殊化查找函数能够采用 char\* 来代替 char\* & 作为第一个参数，但是为了保证推导规则能够正确运行，最好能够保持参数对于非特殊化函数与特殊化函数是一致的。

可以这样使用特殊化：

```
char* word = "two";
char* arr[4] = {"one", "two", "three", "four"};
int res;

res = Find<char*>(word, arr, 4); // Calls the char* specialization
res = Find(word, arr, 4); // Calls the char* specialization
```

### 11.3.2 函数模板的重载

也可以使用非模板函数来重载模板函数。例如，可以编写一个非模板函数 Find() 处理 char\*，来取代针对 char\* 的 Find() 模板特殊化。

```
int Find(char*& value, char** arr, int size)
{
    for (int i = 0; i < size; i++) {
        if (strcmp(arr[i], value) == 0) {
            // Found it; return the index
            return (i);
        }
    }
    // Failed to find it; return -1
    return (-1);
}
```

这个函数在行为上和前一小节所见的特殊化版本是相同的。但是，确定何时调用该函数的规则有所不同：

```
char* word = "two";
char* arr[4] = {"one", "two", "three", "four"};
int res;

res = Find<char*>(word, arr, 4); // Calls the Find template with T=char*
res = Find(word, arr, 4); // Calls the Find nontemplate function!
```

因此，如果希望函数总能正常运行，即不仅在明确指定 char\* 时能够良好运行，在没有明确指定类型时通过推导也能很好地运行，就应该编写模板特殊化，而不是建立重载版本的非模板函数。

像模板类方法定义一样，函数模板定义（不仅仅是原型）必须对使用它们的所有源文件都是可用的。因此，如果不只一个源文件用到函数模板定义，就应该把这些定义放在头文件中。

#### 同时重载和特殊化函数模板

有这种可能：针对 char\* 编写了一个特殊化 Find() 模板，同时还为 char\* 编写了一个独立的 Find() 函数。与模板化函数相比，编译器更喜欢非模板函数。但是，如果明确指定了模板实例化，编译器会被迫使用模板：

```
char* word = "two";
char* arr[4] = {"one", "two", "three", "four"};
int res;

res = Find<char*>(word, arr, 4); // Calls the char* specialization of the
                                // template
res = Find(word, arr, 4); // Calls the Find nontemplate function.
```

### 11.3.3 类模板的友元函数模板

需要在类模板中重载操作符时，函数模板很有用。例如，你可能要为 Grid 类模板重载插入操作符将网格作为流来处理。

如果不熟悉如何重载操作符 `operator<<`，详细的信息请参阅第 16 章。

就像第 16 章讨论的那样，不能让 `operator<<` 成为类 Grid 的一个方法，它必须是一个独立的函数模板。这个定义应该直接放在 Grid.h 中，具体如下：

```
template <typename T>
ostream& operator<<(ostream& ostr, const Grid<T>& grid)
{
    for (int i = 0; i < grid.mHeight; i++) {
        for (int j = 0; j < grid.mWidth; j++) {
            // Add a tab between each element of a row.
            ostr << grid.mCells[j][i] << "\t";
        }
        ostr << std::endl; // Add a newline between each row.
    }
    return (ostr);
}
```

只要存在针对网格元素的插入操作符，这个函数模板就能用于任何 Grid。惟一的问题是：`operator<<` 要访问类 Grid 的 protected 方法。因此，它必须是类 Grid 的友元 (friend)。但是，类 Grid 和 `operator<<` 都是模板。而你实际想要的是：针对特定类型 T 的 `operator<<` 的每个实例化都要成为针对该类型的 Grid 模板实例化的友元。语法如下：

```
//Grid.h
#include <iostream>
using std::ostream;

// Forward declare Grid template.
template <typename T> class Grid;

// Prototype for templated operator<<.
template<typename T>
ostream& operator<<(ostream& ostr, const Grid<T>& grid);
template <typename T>
class Grid
{
public:
    // Omitted for brevity
    friend ostream& operator<< <T>(ostream& ostr, const Grid<T>& grid);
    // Omitted for brevity
};
```

这个友元声明有点棘手，这相当于指出，对于针对类型 T 的模板实例化，`operator<<` 的 T 实例化是其友元 (friend)。换句话说，在类实例化和函数实例化之间有一个一对一的友元映射关系。需要特别注意的是 `operator<<` 上明确的模板声明 `<T>` (`operator<<` 之后的空格是可选的)。这就告诉编译器，`operator<<` 本身就是模板。一些编译器不支持这种语法，但是在 C++ 中它是合法的，在大部分新的编译器上可以通过。

## 11.4 高级模板

本章的前半部分涵盖了类模板和函数模板使用最广泛的特性。如果你只对模板的基本知识感兴趣，认为能够使用 STL 或者编写简单类就可以了，那么读到这里就行了。但是，如果你对模板感兴趣，想释放它们的全部力量，请继续阅读本章的第二部分，学习更难懂但是更有意思的一些细节。

### 11.4.1 关于模板参数的更多知识

实际上有三类模板参数：类型模板参数、无类型模板参数和模板模板参数（注意，不是多重复了一遍：名字真是如此！译者注：以模板作为模板参数）。前面已经看到了类型模板参数和无类型模板参数的例子，不过还没有看到过模板模板参数。另外，对于模板模板参数和无类型模板参数，还有一些更错综复杂的方面，这些方面在前面尚未提及。

#### 关于模板类型参数的更多知识

模板的类型参数正是使用模板的主要目的。只要愿意，你可以声明任意多个类型参数。例如，如果要为网格模板增加第二个类型参数，来指定另一个模板化的类容器，在这个容器上来建立网格。回忆一下第 4 章，标准模板库定义了几种模板化的容器类，包括 `vector` 和 `deque`。在原 `grid` 类中，你可能希望有一个向量数组或者双端队列数组，而不是数组的数组。通过使用另一个模板类型参数，就可以让用户指定他希望底层容器是 `vector` 还是 `deque`。下面就是带有两个模板参数的类定义：

```
template <typename T, typename Container>
class Grid
{
public:
    Grid(int inWidth = kDefaultWidth, int inHeight = kDefaultHeight);
    Grid(const Grid<T, Container>& src);
    ~Grid();

    Grid<T, Container>& operator=(const Grid<T, Container>& rhs);
    void setElementAt(int x, int y, const T& inElem);
    T& getElementAt(int x, int y);
    const T& getElementAt(int x, int y) const;
    int getHeight() const { return mHeight; }
    int getWidth() const { return mWidth; }
    static const int kDefaultWidth = 10;
    static const int kDefaultHeight = 10;

protected:
    void copyFrom(const Grid<T, Container>& src);
    Container* mCells;
    int mWidth, mHeight;
};
```

现在这个模板有两个参数了：T 和 Container。因此，在前面引用 `Grid<T>` 的地方现在必须引用 `Grid<T, Container>` 指定这两个模板参数。另一个变化是，`mCells` 现在使用指向动态分配的 Container 数组的指针，而不是指向动态分配的 T 元素二维数组。

下面是构造函数的定义。在此假设 Container 类型包括 `resize()` 方法。如果你试图指定没有 `resize()` 方法的类型来实例化这个模板，编译器会产生一个错误，如下所述。

```
template <typename T, typename Container>
Grid<T, Container>::Grid(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
```



```
// Dynamically allocate the array of mWidth containers
mCells = new Container[mWidth];
for (int i = 0; i < mWidth; i++) {
    // Resize each container so that it can hold mHeight elements.
    mCells[i].resize(mHeight);
}
}
```

下面是析构函数的定义。在构造函数中只调用过一次 new，所以在析构函数中也只调用一次 delete。

```
template <typename T, typename Container>
Grid<T, Container>::~~Grid()
{
    delete [] mCells;
}
```

copyFrom()的代码假定，可以使用数组[]记法访问容器中的元素。第 16 章解释了如何重载[]操作符，从而在自己的容器类中实现这个特性，但是现在仅仅知道 STL 中的 vector 和 deque 都支持这个语法就可以了。

```
template <typename T, typename Container>
void Grid<T, Container>::copyFrom(const Grid<T, Container>& src)
{
    int i, j;
    mWidth = src.mWidth;
    mHeight = src.mHeight;
    mCells = new Container[mWidth];
    for (i = 0; i < mWidth; i++) {
        // Resize each element, as in the constructor.
        mCells[i].resize(mHeight);
    }
    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}
```

下面是其他方法的实现。

```
template <typename T, typename Container>
Grid<T, Container>::Grid(const Grid<T, Container>& src)
{
    copyFrom(src);
}
```

```
template <typename T, typename Container>
Grid<T, Container>& Grid<T, Container>::operator=(const Grid<T, Container>& rhs)
{
    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }
    // Free the old memory.
    delete [] mCells;
```



```

        // Copy the new memory.
        copyFrom(rhs);

        return (*this);
    }

template <typename T, typename Container>
void Grid<T, Container>::setElementAt(int x, int y, const T& inElem)
{
    mCells[x][y] = inElem;
}

template <typename T, typename Container>
T& Grid<T, Container>::getElementAt(int x, int y)
{
    return (mCells[x][y]);
}

template <typename T, typename Container>
const T& Grid<T, Container>::getElementAt(int x, int y) const
{
    return (mCells[x][y]);
}

```

可以如下实例化并使用网格对象：

```

Grid<int, vector<int> > myIntGrid;
Grid<int, deque<int> > myIntGrid2;

myIntGrid.setElementAt(3, 4, 5);
cout << myIntGrid.getElementAt(3, 4);

Grid<int, vector<int> > grid2(myIntGrid);
grid2 = myIntGrid;

```

参数名使用 Container 一词并不表示该类型就真的必须是容器。可以试着用 int 实例化类 Grid：

```
Grid<int, int> test; // WILL NOT COMPILE
```

这一行代码不能编译通过，但是编译器报告的错误可能不是你想象的错误。它不会解释第二个类型参数是用 int 代替了 Container。它会告诉你：left of ‘.resize’ must have class/struct/union type (‘.resize’ 的左边必须有 class/struct/union 类型)。这是因为编译器是企图用 int 作为 Container 来生成 Grid 类。在编译器编译到这一行之前，所有的代码都能正常运行：

```
mCells[i].resize(mHeight);
```

在这一点上，编译器会意识到 mCells[i] 是 int，所以不能对它调用 resize() 方法！

这个方法看起来好像走了一些弯路，而且感觉对你也没有什么用处。但是，在标准模板库中确实采用了这种方法。stack、queue 和 priority\_queue 等类模板都采用了模板类型参数来指定底层容器，这些容器可以是 vector、deque 或者 list。

#### 模板类型参数的默认值

可以为模板参数设置默认值。例如，你可能想说，你的 Grid 的默认容器是 vector。模板类的定义

如下：

```
#include <vector>
using std::vector;

template <typename T, typename Container = vector<T> >
class Grid
{
public:
    // Everything else is the same as before.
};
```

可以使用第一个模板参数类型 T 作为第二个模板参数默认值中 vector 模板的参数。还要注意一点，就是必须在两个相邻的尖括号之间保留一个空格来避免这一章前面讨论的解析问题。

C++ 语法要求，不能在方法定义的模板头所在的这行代码中重复默认值。有了这个默认值，客户代码可以指定底层容器，也可以不指定底层容器。

```
Grid<int, vector<int> > myIntGrid;
Grid<int> myIntGrid2;
```

### 引入模板模板参数

前面的 Container 参数存在一个问题。在实例化这个类模板时，会使用这样的代码：

```
Grid<int, vector<int> > myIntGrid;
```

注意，其中的 int 类型出现了两次。必须指定 Grid 和 vector 的元素类型都是 int。如果这样写会怎么样呢？

```
Grid<int, vector<SpreadsheetCell> > myIntGrid;
```

它就会运行不正常！像下面这样写似乎应该可以，这样就不会犯上面的错误了。

```
Grid<int, vector> myIntGrid;
```

Grid 类应该能够得出它需要一个 int 元素组成的 vector。不过，编译器不允许你将这个实参传递给正常的类型参数，因为 vector 本身并不是类型，而是一个模板。

如果需要使用模板作为模板参数，必须使用一种特定类型的参数，称为模板模板参数（template template parameter）。它的语法有点令人抓狂，一些编译器还不支持模板模板参数。如果你对它还很有兴趣，请往下继续阅读。

指定模板模板参数有点像在常规的函数中指定函数指针参数。函数指针类型包括返回类型和函数的参数类型。类似的，在指定模板模板参数时，模板模板参数的完整规范要包括该模板的参数。

STL 中的容器包含一个模板参数列表，形式如下：

```
template <typename E, typename Allocator = allocator<E> >
class vector
{
    // Vector definition
};
```

参数 E 只是元素类型。现在不要担心 Allocator 的问题——有关内容将在第 21 章介绍。

给定上面的模板规范，下面是类 Grid 的类模板定义，它取容器模板作为第二个模板参数：

```
template <typename T, template <typename E, typename Allocator = allocator<E> >
class Container = vector >
class Grid
{
    public:
        // Omitted code that is the same as before
        Container<T>* mCells;
        // Omitted code that is the same as before
};
```

这里有什么情况呢？第一个模板参数与前面的相同，也是元素类型 T。第二个模板参数现在是像 vector 或 deque 这样的容器模板本身。就像前面看到的那样，这个“模板类型”必须取两个参数：元素类型 E 和分配器 Allocator。注意，在嵌套模板参数列表之后 class 一词出现了两次。模板 Grid 中的这个参数的名字是 Container（和前面一样）。现在的默认值是 vector，而不是 vector<T>，因为 Container 是一个模板，而不是实际类型。

模板模板参数的一般语法规则如下：

```
template <other params, ..., template <TemplateTypeParams> class ParameterName,
other params, ...>
```

现在你已经领略了上面声明模板的语法有多麻烦，其余的部分就容易了。必须指定 Container<T> 作为你使用的容器类型，而不是在代码中使用 Container 本身。例如，现在的构造函数如下（在定义方法的模板规范中，不用重复默认的模板模板参数实参）。

```
template <typename T, template <typename E, typename Allocator = allocator<E> >
class Container>
Grid<T, Container>::Grid(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    mCells = new Container<T>[mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i].resize(mHeight);
    }
}
```

实现了全部方法之后，可以这样来使用模板：

```
Grid<int, vector> myGrid;

myGrid.setElementAt(1, 2, 3);
myGrid.getElementAt(1, 2);
Grid<int, vector> myGrid2(myGrid);
```

如果你没有完全跳过这一节，而是认真地阅读了上述内容，现在你肯定认为，C++ 遭到的谴责确实是有来由的。不要陷于这里的语法泥沼中，在心里谨记一个核心概念就行了，这就是可以把模板作为参数传递给其他模板。

#### 关于无类型模板参数的更多知识

你可能想要允许用户指定一个（不是字面意义上的）空元素，用来初始化网格中的每个单元格。有一个很完美的合理方法来实现这个目标：

```
template <typename T, const T EMPTY>
class Grid
{
public:
    Grid(int inWidth = kDefaultWidth, int inHeight = kDefaultHeight);
    Grid(const Grid<T, EMPTY>& src);
    ~Grid();
    Grid<T, EMPTY>& operator=(const Grid<T, EMPTY>& rhs);

    // Omitted for brevity

protected:
    void copyFrom(const Grid<T, EMPTY>& src);
    T** mCells;
    int mWidth, mHeight;
};
```

这个定义是合法的。可以使用来自第一个参数的类型 `T` 作为第二个参数的类型，无类型参数可以用 `const` 声明，就像函数参数一样。可以使用这个 `T` 的初始值来初始化网格中的每个单元格：

```
template <typename T, const T EMPTY>
Grid<T, EMPTY>::Grid(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    mCells = new T* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new T[mHeight];
        for (int j = 0; j < mHeight; j++) {
            mCells[i][j] = EMPTY;
        }
    }
}
```

另一个方法的定义保持不变，除了必须在模板行增加第二个类型参数之外，`Grid<T>` 的所有实例变成了 `Grid<T, EMPTY>`。在进行这些修改之后，就可以用一个初始值来实例化 `int Grid` 的所有元素了：

```
Grid<int, 0> myIntGrid;
Grid<int, 10> myIntGrid2;
```

初始值可以是任意整数。但是，如果试图创建一个 `SpreadsheetCell Grid`：

```
SpreadsheetCell emptyCell;
Grid<SpreadsheetCell, emptyCell> mySpreadsheet; // WILL NOT COMPILE
```

这一行会导致编译器出现错误，因为不能把对象作为实参传递给无类型参数。

无类型参数不能是对象，甚至不能是 `double` 或者 `float`。无类型参数仅限于 `int`、`enum`、指针和引用。

这个例子说明了模板类的一个特殊情况：在一种类型上正确运行的模板类对另一种类型可能无法编译通过。

#### 引用和指针无类型模板参数

要允许用户为网格指定一个初始空元素，一个更妥善的方法是：使用 `T` 的一个引用作为无类型模板

参数。下面是新的类定义：

```
template <typename T, const T& EMPTY>
class Grid
{
    // Everything else is the same as the previous example, except the
    // template lines in the method definitions specify const T& EMPTY
    // instead of const T EMPTY.
};
```

现在可以针对任意类型实例化这个模板类了。但是，作为第二个模板参数传递的引用必须指向一个带有外部连接（external linkage）的全局变量。外部连接可以看作与静态（static）连接相对，这表示变量在定义它的源文件之外的其他源文件中是可用的。关于更多细节请参阅第 12 章。现在，你只要知道可以用关键字 `extern` 声明变量有外部连接就可以了：

```
extern const int x = 0;
```

注意，这一行出现在任何函数或方法体之外。这里有一个完整的程序，它声明了带有初始化参数的 `int` 和 `SpreadsheetCell` 网格。

```
#include "GridRefNonType.h"
#include "SpreadsheetCell.h"

extern const int emptyInt = 0;
extern const SpreadsheetCell emptyCell(0);

int main(int argc, char** argv)
{
    Grid<int, emptyInt> myIntGrid;
    Grid<SpreadsheetCell, emptyCell> mySpreadsheet;

    Grid<int, emptyInt> myIntGrid2(myIntGrid);

    return (0);
}
```

引用和指针模板参数必须指向所有翻译单元中都可用的全局变量。对于这些类型的变量，相应的技术术语就是带有外部连接的数据。

### 使用 0 初始化模板类型

到目前为止，给单元格提供初始空值的两种方法都不是太完美。你可能只是想把每个单元格初始化为你选择的一个合理的默认值（而不是让用户指定）。当然，这会带来一个直接的问题：对于每一种可能的类型，什么值才算合理呢？对于对象，合理的值就是用默认构造函数创建的对象。实际上，创建对象数组时所得到的正是这种默认对象。然而，对于像 `int` 和 `double` 这样的简单数据类型以及指针来说，合理的初始值则是 0。因此，真正要做的只是将非对象类型赋值为 0，对于对象则使用默认构造函数。在 11.2.5 节已经看到过这种语法。下面是使用 0 初始化语法实现 `Grid` 模板构造函数的代码：

```
template <typename T>
Grid<T>::Grid(int inWidth, int inHeight) : mWidth(inWidth), mHeight(inHeight)
{
    mCells = new T* [mWidth];
    for (int i = 0; i < mWidth; i++) {
```



```

        mCells[i] = new T[mHeight];
        for (int j = 0; j < mHeight; j++) {
            mCells[i][j] = T();
        }
    }
}

```

基于此，你能够还原原来的 Grid 类（不带 EMPTY 无类型参数），并且只是把每个单元格元素初始化为 0 初始化“合理值”。

#### 11.4.2 模板类的部分特殊化

本章的第一部分给出的 char\* 类特殊化称为类模板完全特殊化（full class template specialization），因为它对 Grid 模板特殊化时对所有的模板参数都进行了特殊化。在这个特殊化中，再没有模板参数了。这不是对类特殊化的惟一方法；还可以对类编写一个部分特殊化（partial class specialization），即只特殊化模板的某些参数，其他的模型参数则不进行特殊化。比如，回忆一下基本 Grid 模板，它带有宽度和高度无类型参数：

```

template <typename T, int WIDTH, int HEIGHT>
class Grid
{
public:
    void setElementAt(int x, int y, const T& inElem);
    T& getElementAt(int x, int y);
    const T& getElementAt(int x, int y) const;
    int getHeight() const { return HEIGHT; }
    int getWidth() const { return WIDTH; }

protected:
    T mCells[WIDTH][HEIGHT];
};

```

可以像下面这样针对 C 风格的字符串 char\* 对这个模板类特殊化：

```

#include "Grid.h" // The file containing the Grid template definition shown above
#include <cstdlib>
#include <cstring>
using namespace std;

template <int WIDTH, int HEIGHT>
class Grid<char*, WIDTH, HEIGHT>
{
public:
    Grid();
    Grid(const Grid<char*, WIDTH, HEIGHT>& src);
    ~Grid();

    Grid<char*, WIDTH, HEIGHT>& Grid<char*, WIDTH, HEIGHT>::operator=(
        const Grid<char*, WIDTH, HEIGHT>& rhs);
    void setElementAt(int x, int y, const char* inElem);
    char* getElementAt(int x, int y) const;
    int getHeight() const { return HEIGHT; }
    int getWidth() const { return WIDTH; }

protected:
    void copyFrom(const Grid<char*, WIDTH, HEIGHT>& src);
    char* mCells[WIDTH][HEIGHT];
};

```



在这种情况下，并没有特殊化所有的模板参数。因此，模板代码行如下：

```
template <int WIDTH, int HEIGHT>
class Grid<char*, WIDTH, HEIGHT>
```

注意，这个模板只有两个参数：WIDTH 和 HEIGHT。可是，你编写的 Grid 类却有三个参数：T、WIDTH、HEIGHT。因此，你的模板参数列表包含了两个参数，而显式的 Grid<char\*, WIDTH, HEIGHT> 包含了三个参数。在实例化该模板时，就必须指定三个参数。不能只使用高度和宽度实例化这个模板：

```
Grid<int, 2, 2> myIntGrid; // Uses the original Grid
Grid<char*, 2, 2> myStringGrid; // Uses the partial specialization for char *s
Grid<2, 3> test; // DOES NOT COMPILE! No type specified.
```

的确，语法是比较混乱，而且还可能会变得更糟。不像完全特殊化，对于部分特殊化，要在所有的方法定义之前包含模板行：

```
template <int WIDTH, int HEIGHT>
Grid<char*, WIDTH, HEIGHT>::Grid()
{
    for (int i = 0; i < WIDTH; i++) {
        for (int j = 0; j < HEIGHT; j++) {
            // Initialize each element to NULL.
            mCells[i][j] = NULL;
        }
    }
}
```

这个模板行要有两个参数来说明这个方法是针对这两个参数来参数化的。注意，不管在哪里引用完全类名，都必须使用 Grid<char\*, WIDTH, HEIGHT>。

其他的方法定义如下：

```
template <int WIDTH, int HEIGHT>
Grid<char*, WIDTH, HEIGHT>::Grid(const Grid<char*, WIDTH, HEIGHT>& src)
{
    copyFrom(src);
}

template <int WIDTH, int HEIGHT>
Grid<char*, WIDTH, HEIGHT>::~Grid()
{
    for (int i = 0; i < WIDTH; i++) {
        for (int j = 0; j < HEIGHT; j++) {
            delete [] mCells[i][j];
        }
    }
}

template <int WIDTH, int HEIGHT>
void Grid<char*, WIDTH, HEIGHT>::copyFrom(
    const Grid<char*, WIDTH, HEIGHT>& src)
{
    int i, j;
```

```

    for (i = 0; i < WIDTH; i++) {
        for (j = 0; j < HEIGHT; j++) {
            if (src.mCells[i][j] == NULL) {
                mCells[i][j] = NULL;
            } else {
                mCells[i][j] = new char[strlen(src.mCells[i][j]) + 1];
                strcpy(mCells[i][j], src.mCells[i][j]);
            }
        }
    }
}

template <int WIDTH, int HEIGHT>
Grid<char*, WIDTH, HEIGHT>& Grid<char*, WIDTH, HEIGHT>::operator=(
    const Grid<char*, WIDTH, HEIGHT>& rhs)
{
    int i, j;

    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }

    // Free the old memory.
    for (i = 0; i < WIDTH; i++) {
        for (j = 0; j < HEIGHT; j++) {
            delete [] mCells[i][j];
        }
    }

    // Copy the new memory.
    copyFrom(rhs);
    return (*this);
}

template <int WIDTH, int HEIGHT>
void Grid<char*, WIDTH, HEIGHT>::setElementAt(
    int x, int y, const char* inElem)
{
    delete[] mCells[x][y];
    if (inElem == NULL) {
        mCells[x][y] = NULL;
    } else {
        mCells[x][y] = new char[strlen(inElem) + 1];
        strcpy(mCells[x][y], inElem);
    }
}

template <int WIDTH, int HEIGHT>
char* Grid<char*, WIDTH, HEIGHT>::getElementAt(int x, int y) const
{
    if (mCells[x][y] == NULL) {
        return (NULL);
    }

    char* ret = new char[strlen(mCells[x][y]) + 1];
    strcpy(ret, mCells[x][y]);

    return (ret);
}

```

### 部分特殊化的另一种形式

上一个例子并没有展示出部分特殊化的真正功能。可以给部分可能的类型（只是所有类型的一个子集）编写特殊化实现，而不是各个类型都进行特殊化。例如，可以针对所有指针类型编写 Grid 类的特殊化。这个特殊化可以完成指针所指对象的深复制，而不是在网格中存储指针的浅复制。

下面是这个类的定义，在此假定特殊化只有一个参数的 Grid 模板。

```
#include "Grid.h"

template <typename T>
class Grid<T*>
{
public:
    Grid(int inWidth = kDefaultWidth, int inHeight = kDefaultHeight);
    Grid(const Grid<T*>& src);
    ~Grid();
    Grid<T*>& operator=(const Grid<T*>& rhs);

    void setElementAt(int x, int y, const T* inElem);
    T* getElementAt(int x, int y) const;
    int getHeight() const { return mHeight; }
    int getWidth() const { return mWidth; }
    static const int kDefaultWidth = 10;
    static const int kDefaultHeight = 10;

protected:
    void copyFrom(const Grid<T*>& src);
    T** mCells;
    int mWidth, mHeight;
};
```

就像平常一样，以下这两行是关键：

```
template <typename T>
class Grid<T*>
```

以上语法指出，这个类是 Grid 模板针对所有指针类型的特殊化。最起码编译器所了解的就是这样。不过对于我们来说，从这可以看出，C++ 标准委员应该提出一个更好的语法才对！除非你已经使用这种语法很长时间了，否则就会发现这是很不合适的。

只有当 T 是指针类型时才会提供特殊化实现。需要注意的是，如果像这样实例化一个网格：Grid<int\*>myIntGrid，那么 T 实际上将是 int 而不是 int\*。这就有点不太直观，但是遗憾的是，它的确就是这样工作的。下面是一个示例代码：

```
Grid<int*> psGrid(2, 2); // Uses the partial specialization for pointer types

int x = 3, y = 4;
psGrid.setElementAt(0, 0, &x);
psGrid.setElementAt(0, 1, &y);
psGrid.setElementAt(1, 0, &y);
psGrid.setElementAt(1, 1, &x);

Grid<int> myIntGrid; // Uses the nonspecialized grid
```

此时，你可能不太清楚这是否能够真正工作。有这种怀疑是正常的。作者之一第一次遇到这个语法的时候也感到很吃惊，在进行实际的实验之前他也不相信这真的能正常工作。如果不相信我们说的，可

以亲自实验一下！以下是方法的实现。要特别注意每个方法之前的模板行的语法。

```
template <typename T>
const int Grid<T*>::kDefaultWidth;

template <typename T>
const int Grid<T*>::kDefaultHeight;

template <typename T>
Grid<T*>::Grid(int inWidth, int inHeight) : mWidth(inWidth), mHeight(inHeight)
{
    mCells = new T* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new T[mHeight];
    }
}

template <typename T>
Grid<T*>::Grid(const Grid<T*>& src)
{
    copyFrom(src);
}

template <typename T>
Grid<T*>::~~Grid()
{
    // Free the old memory.
    for (int i = 0; i < mWidth; i++) {
        delete [] mCells[i];
    }
    delete [] mCells;
}

template <typename T>
void Grid<T*>::copyFrom(const Grid<T*>& src)
{
    int i, j;
    mWidth = src.mWidth;
    mHeight = src.mHeight;

    mCells = new T* [mWidth];
    for (i = 0; i < mWidth; i++) {
        mCells[i] = new T[mHeight];
    }

    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}

template <typename T>
Grid<T*>& Grid<T*>::operator=(const Grid<T*>& rhs)
{
    // Check for self-assignment.
```

```

    if (this == &rhs) {
        return (*this);
    }
    // Free the old memory.
    for (int i = 0; i < mWidth; i++) {
        delete [] mCells[i];
    }
    delete [] mCells;

    // Copy the new memory.
    copyFrom(rhs);
    return (*this);
}

template <typename T>
void Grid<T*>::setElementAt(int x, int y, const T* inElem)
{
    mCells[x][y] = *inElem;
}

template <typename T>
T* Grid<T*>::getElementAt(int x, int y) const
{
    T* newElem = new T(mCells[x][y]);
    return (newElem);
}

```

### 11.4.3 用重载模板函数部分特殊化

C++ 标准不允许对函数进行部分模板特殊化。但是可以使用另一个模板来重载函数。其中的区别比较微妙。假设你要编写本章前面给出的函数 Find() 的特殊化，这个函数要对指针解除引用，从而对指针所指的對象直接使用 operator== 的指针。按照类模板部分特殊化的语法，可能会编写如下代码：

```

template <typename T>
int Find<T*>(T*& value, T** arr, int size)
{
    for (int i = 0; i < size; i++) {
        if (*arr[i] == *value) {
            // Found it; return the index
            return (i);
        }
    }
    // Failed to Find it; return -1
    return (-1);
}

```

然而，这个语法声明了函数模板的一个部分特殊化，这是 C++ 标准不允许的（虽然有一些编译器确实支持这个语法）。要实现你需要的行为，标准方法是为函数 Find() 编写一个新的模板：

```

template <typename T>
int Find(T*& value, T** arr, int size)
{
    for (int i = 0; i < size; i++) {
        if (*arr[i] == *value) {
            // Found it; return the index

```



```
        return (i);
    }
}
// Failed to Find it; return -1
return (-1);
}
```

看起来这里的区别没有什么，好像只是纸上文章，没有实际意义，这样的代码将是可移植的标准代码，而前面的做法则可能得到无法编译通过的代码。

#### 关于类型推导的更多知识

可以在一个程序中定义多个 Find()，包括定义初始的 Find() 模板，为对应指针类型的部分特殊化定义重载的 Find()，对应 char\* 定义完全特殊化，并只为 char\* 定义重载的 Find()。编译器会在它的推导规则上选择正确的版本调用。

编译器总是选择“最特定”的函数，非模板函数总是比模板函数更优先。

下面这段代码调用了指定“版本”的 Find() 函数：

```
char* word = "two";
char* arr[4] = {"one", "two", "three", "four"};
int res;

int x = 3, intArr[4] = {1, 2, 3, 4};
double d1 = 5.6, dArr[4] = {1.2, 3.4, 5.7, 7.5};

res = Find(x, intArr, 4); // Calls Find<int> by deduction
res = Find<int>(x, intArr, 4); // Call Find<int> explicitly

res = Find(d1, dArr, 4); // Call Find<double> by deduction
res = Find<double>(d1, dArr, 4); // Calls Find<double> explicitly

res = Find<char*>(word, arr, 4); // Calls template specialization for char*s
res = Find(word, arr, 4); // Calls the overloaded Find for char*s

int *px = &x, *pArr[2] = {&x, &x};
res = Find(px, pArr, 2); // Calls the overloaded Find for pointers

SpreadsheetCell c1(10), c2[2] = {SpreadsheetCell(4), SpreadsheetCell(10)};

res = Find(c1, c2, 2); // Calls Find<SpreadsheetCell> by deduction
res = Find<SpreadsheetCell>(c1, c2, 2); // Calls Find<SpreadsheetCell>
// explicitly

SpreadsheetCell *pc1 = &c1;
SpreadsheetCell *psa[2] = {&c1, &c1};

res = Find(pc1, psa, 2); // Calls the overloaded Find for pointers
```

#### 11.4.4 模板递归

C++ 中的模板提供了一些强大的能力，这远非本章中到目前为止你所见过的简单类和函数所能做到的。其中一种功能是模板递归 (template recursion)。本节首先介绍引入模板递归的初衷，然后说明如何实现模板递归。



本节采用了第 16 章讨论的一些操作符重载特性。如果你不熟悉重载 operator[] 的语法，请参考第 16 章。

### N 维网格：第一次尝试

本章前面的 Grid 模板的例子仅仅支持二维，这就限制了它的用途。如果需要编写 3-D 的 Tic-Tac-Toe 游戏或者编写四维矩阵的数学程序该怎么办呢？当然，你可以给每一维都编写一个模板或非模板类。但这样做就会重复很多代码。另一种方法是只编写一维的网格。然后，使用一种 Grid 作为另一种 Grid 的元素类型来实例化后者，从而创建任意维数的 Grid。这个 Grid 元素类型自己也可以用一个 Grid 作为元素类型来实例化，依此类推。下面是 OneDGrid 类模板的实现。它只是前面的 Grid 模板的一维实例，只是额外增加了 resize() 方法，并用 operator [] 取代了 setElementAt() 和 getElementAt()。当然，成品代码中对数组访问完成越界检查，如果某些地方有问题的话应抛出异常。

```
template <typename T>
class OneDGrid
{
public:
    OneDGrid(int inSize = kDefaultSize);
    OneDGrid(const OneDGrid<T>& src);
    ~OneDGrid();

    OneDGrid<T> &operator=(const OneDGrid<T>& rhs);
    void resize(int newSize);

    T& operator[](int x);
    const T& operator[](int x) const;
    int getSize() const { return mSize; }
    static const int kDefaultSize = 10;
protected:
    void copyFrom(const OneDGrid<T>& src);
    T* mElems;
    int mSize;
};
```

```
template <typename T>
const int OneDGrid<T>::kDefaultSize;
```

```
template <typename T>
OneDGrid<T>::OneDGrid(int inSize) : mSize(inSize)
{
    mElems = new T[mSize];
}
```

```
template <typename T>
OneDGrid<T>::OneDGrid(const OneDGrid<T>& src)
{
    copyFrom(src);
}
```

```
template <typename T>
OneDGrid<T>::~~OneDGrid()
{
    delete [] mElems;
}
```

```

template <typename T>
void OneDGrid<T>::copyFrom(const OneDGrid<T>& src)
{
    mSize = src.mSize;
    mElems = new T[mSize];

    for (int i = 0; i < mSize; i++) {
        mElems[i] = src.mElems[i];
    }
}

template <typename T>
OneDGrid<T>& OneDGrid<T>::operator=(const OneDGrid<T>& rhs)
{
    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }

    // Free the old memory.
    delete [] mElems;

    // Copy the new memory.
    copyFrom(rhs);
    return (*this);
}

template <typename T>
void OneDGrid<T>::resize(int newSize)
{
    T* newElems = new T[newSize]; // Allocate the new array of the new size

    // Handle the new size being smaller or bigger than the old size.
    for (int i = 0; i < newSize && i < mSize; i++) {
        // Copy the elements from the old array to the new one.
        newElems[i] = mElems[i];
    }
    mSize = newSize; // Store the new size.
    delete [] mElems; // Free the memory for the old array.
    mElems = newElems; // Store the pointer to the new array.
}

template <typename T>
T& OneDGrid<T>::operator[](int x)
{
    return (mElems[x]);
}

template <typename T>
const T& OneDGrid<T>::operator[](int x) const
{
    return (mElems[x]);
}

```

有了这个 OneDGrid 的实现，就可以创建多维网格了。

```

OneDGrid<int> singleDGrid;
OneDGrid<OneDGrid<int> > twoDGrid;
OneDGrid<OneDGrid<OneDGrid<int> > > threeDGrid;

singleDGrid[3] = 5;
twoDGrid[3][3] = 5;
threeDGrid[3][3][3] = 5;

```

这段代码能够很好地运行，但是它的声明很乱。我们还可以做得更好。

### 真正的 N 维网格

可以使用模板递归来编写“真正的”N 维网格，因为网格的维数本质上就是递归的。从下面这个声明可以看到这一点：

```
OneDGrid< OneDGrid< OneDGrid< int> > > threeDGrid;
```

可以把每个嵌套 OneDGrid 看作一个递归步骤，以 int 的 OneDGrid 作为基本情况（基例）。换句话说，三维网格就是 int 的一维网格的一维网格的一维网格。你可以编写一个模板类来实现该功能，而不是要求用户完成这样的递归。然后，可以如下创建 N 维网格：

```

NDGrid<int, 1> singleDGrid;
NDGrid<int, 2> twoDGrid;
NDGrid<int, 3> threeDGrid;

```

NDGrid 模板类为其元素取一个类型，并用一个整数指定“维数”。这里关键的一点是要了解到，NDGrid 的元素类型不是在模板参数列表中指定的元素类型，而实际上是比当前维数小一维的另一个 NDGrid。换句话说，三维网格是二维网格的数组；这些二维网格又是一维网格的数组。

使用递归时需要一个基本情况（基例）。可以给维数为 1 的网格编写 NDGrid 的一个部分特殊化，其中的元素类型不是另一个 NDGrid，而确实是模板参数指定的元素类型。

以下是一个通用的 NDGrid 模板定义，在此突出显示了它与上面给出的 OneDGrid 之间的区别。

```

template <typename T, int N>
class NDGrid
{
public:
    NDGrid();
    NDGrid(int inSize);
    NDGrid(const NDGrid<T, N>& src);
    ~NDGrid();

    NDGrid<T, N>& operator=(const NDGrid<T, N>& rhs);
    void resize(int newSize);
    NDGrid<T, N-1>& operator[](int x);
    const NDGrid<T, N-1>& operator[](int x) const;
    int getSize() const { return mSize; }
    static const int kDefaultSize = 10;
protected:
    void copyFrom(const NDGrid<T, N>& src);
    NDGrid<T, N-1>* mElems;
    int mSize;
};

```

注意，mElems 是指向 NDGrid<T, N-1> 的指针；这是递归步骤。另外，operator[] 返回元素类型

的引用, 这也是 `NDGrid<T, N-1>`, 而不是 `T`。

下面是基本情况 (基例) 的模板定义:

```
template <typename T>
class NDGrid<T, 1>
{
public:
    NDGrid(int inSize = kDefaultSize);
    NDGrid(const NDGrid<T, 1>& src);
    ~NDGrid();
    NDGrid<T, 1>& operator=(const NDGrid<T, 1>& rhs);
    void resize(int newSize);
    T& operator[](int x);
    const T& operator[](int x) const;
    int getSize() const { return mSize; }
    static const int kDefaultSize = 10;
protected:
    void copyFrom(const NDGrid<T, 1>& src);
    T* mElems;
    int mSize;
};
```

这是递归的终点: 元素类型是 `T`, 而不是另一个模板实例化。

实现中最棘手的方面除了模板递归自身以外, 就是正确建立数组的每一维的大小。这个实现创建了一个 `N` 维数组, 每一维的大小都相等。为每一维指定不同的大小更加困难。但是, 即使做了上述简化, 这里还存在一个问题: 应该能够让用户指定数组的大小, 比如 20 或 50。这样, 要有一个构造函数取一个大小参数 (整数)。但是, 在动态分配网格的嵌套数组时, 不能把数组大小值传递给网格, 因为数组是使用对象的默认构造函数来创建对象的。因此, 必须在数组的每个网格元素上显式地调用 `resize()`。代码如下所示, 为了清晰起见, 默认构造函数和有一个参数的构造函数是分开显示的。

对于基本情况 (基例) 来说, 因为元素是 `T`, 不是网格, 所以不需要改变元素的大小。

下面是主 `NDGrid` 模板的实现, 在此突出显示了它与 `OneDGrid` 的区别:

```
template <typename T, int N>
const int NDGrid<T, N>::kDefaultSize;
```

```
template <typename T, int N>
NDGrid<T, N>::NDGrid(int inSize) : mSize(inSize)
```

```
{
    mElems = new NDGrid<T, N-1>[mSize];
    // Allocating the array above calls the 0-argument
    // constructor for the NDGrid<T, N-1>, which constructs
    // it with the default size. Thus, we must explicitly call
    // resize() on each of the elements.
    for (int i = 0; i < mSize; i++) {
        mElems[i].resize(inSize);
    }
}
```

```
template <typename T, int N>
NDGrid<T, N>::NDGrid() : mSize(kDefaultSize)
{
    mElems = new NDGrid<T, N-1>[mSize];
}
```

```
template <typename T, int N>
NDGrid<T, N>::NDGrid(const NDGrid<T, N>& src)
{
    copyFrom(src);
}
```

```
template <typename T, int N>
NDGrid<T, N>::~~NDGrid()
{
    delete [] mElems;
}
```

```
template <typename T, int N>
void NDGrid<T, N>::copyFrom(const NDGrid<T, N>& src)
{
    mSize = src.mSize;
    mElems = new NDGrid<T, N-1>[mSize];
    for (int i = 0; i < mSize; i++) {
        mElems[i] = src.mElems[i];
    }
}
```

```
template <typename T, int N>
NDGrid<T, N>& NDGrid<T, N>::operator=(const NDGrid<T, N>& rhs)
{
    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }
    // Free the old memory.
    delete [] mElems;
    // Copy the new memory.
    copyFrom(rhs);
    return (*this);
}
```

```
template <typename T, int N>
void NDGrid<T, N>::resize(int newSize)
{
    // Allocate the new array with the new size.
    NDGrid<T, N - 1>* newElems = new NDGrid<T, N - 1>[newSize];
    // Copy all the elements, handling the cases where newSize is
    // larger than mSize and smaller than mSize.
    for (int i = 0; i < newSize && i < mSize; i++) {
        newElems[i] = mElems[i];
        // Resize the nested Grid elements recursively.
        newElems[i].resize(newSize);
    }
    // Store the new size and pointer to the new array.
    // Free the memory for the old array first.
    mSize = newSize;
    delete [] mElems;
    mElems = newElems;
}
```

```
template <typename T, int N>
NDGrid<T, N-1>& NDGrid<T, N>::operator[](int x)
```



```
{
    return (mElems[x]);
}
```

```
template <typename T, int N>
const NDGrid<T, N-1>& NDGrid<T, N>::operator[](int x) const
{
    return (mElems[x]);
}
```

下面是部分特殊化（基本情况）的实现。注意，因为没有使用特殊化继承任何实现，所以必须重写许多代码。突出显示的代码展示了它与非特殊化 NDGrid 的区别。

```
template <typename T>
const int NDGrid<T, 1>::kDefaultSize;
```

```
template <typename T>
NDGrid<T, 1>::NDGrid(int inSize) : mSize(inSize)
{
    mElems = new T[mSize];
}
template <typename T>
NDGrid<T, 1>::NDGrid(const NDGrid<T, 1>& src)
{
    copyFrom(src);
}
```

```
template <typename T>
NDGrid<T, 1>::~NDGrid()
{
    delete [] mElems;
}
```

```
template <typename T>
void NDGrid<T, 1>::copyFrom(const NDGrid<T, 1>& src)
{
    mSize = src.mSize;
    mElems = new T[mSize];
    for (int i = 0; i < mSize; i++) {
        mElems[i] = src.mElems[i];
    }
}
```

```
template <typename T>
NDGrid<T, 1>& NDGrid<T, 1>::operator=(const NDGrid<T, 1>& rhs)
{
    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }
    // Free the old memory.
    delete [] mElems;
    // Copy the new memory.
    copyFrom(rhs);
    return (*this);
}
```



```
template <typename T>
void NDGrid<T, 1>::resize(int newSize)
{
    T* newElems = new T[newSize];

    for (int i = 0; i < newSize && i < mSize; i++) {
        newElems[i] = mElems[i];
        // Don't need to resize recursively, because this is the base case.
    }
    mSize = newSize;
    delete [] mElems;
    mElems = newElems;
}
```

```
template <typename T>
T& NDGrid<T, 1>::operator[](int x)
{
    return (mElems[x]);
}

template <typename T>
const T& NDGrid<T, 1>::operator[](int x) const
{
    return (mElems[x]);
}
```

现在可以这样来编写代码：

```
NDGrid<int, 3> my3DGrid;
my3DGrid[2][1][2] = 5;
my3DGrid[1][1][1] = 5;

cout << my3DGrid[2][1][2] << endl;
```

## 11.5 小结

本章的主要内容是使用模板编写通用程序。我们希望你已经为这些特性的力量和功能所折服，并且了解了如何把这些概念应用到自己的代码中。如果第一次阅读的时候没有领会所有的语法，或者不能读懂所有的例子，不要担心。第一次遇到这些概念的时候掌握这些概念是会有些困难，语法确实太复杂了，以至于本书的作者每次需要编写模板的时候也都要查找参考书。当你实际坐下来编写模板类或者函数模板时，可以参考本章来查找正确的语法。

本章主要是为第 21、22、23 章做准备，这三章的主要内容是标准模板库。如果想现在就阅读关于 STL 的知识，可以直接阅读第 21 章～第 23 章，但是我们建议最好还是先阅读第 II 和第 III 部分的其余章节。

## 第12章 理解C++疑难问题

C++程序设计语言有许多错综复杂的语法，语义也很费解。作为C++程序员，你或许已经逐渐熟悉了大部分的内容，开始觉得这样很自然了。但是，C++的某些方面往往很麻烦。要么是书里未能把它们解释透彻，要么是你忘记了怎么来使用，而要反反复复地查找才行，也有可能这两种现象同时存在。本章会清楚地解释C++最让人头疼的一些疑难问题，帮助你消除这样一些困惑。

本书的许多章节中都已经涉及了C++特有的一些语言特征。本章尽量不再重复这些话题，这里的重点是在本书其他章节中未做详细介绍的内容。这些内容与其他章节的内容可能会有一些重叠，但是会采用一种不同的方式进行剖析，使你能够从一个新角度来了解这些内容。

本章的主要内容包括：引用、const、static、extern、typedef、类型强制转换、作用域、头文件、变长参数列表和预处理宏等。虽然列出这些技术看起来像是大杂烩一样，不过要知道，这可是我们精心挑选的，这些技术最令人费解，而它们也是C++语言中最常用的技术。

### 12.1 引用

专业的C++的代码（包括本书中的许多代码）都大量使用了引用。请回想并思考一下什么是引用，引用有怎样的表现，这有助于理解本节的内容。

C++的引用（reference）是另外一个变量的别名（alias）。对引用的所有修改都会改变该引用所指向变量的值。可以把引用看作是一种隐式的指针，它可以免除获取变量地址和对指针解除引用的麻烦。也可以把引用看作是原变量的另一个名字。可以创建独立的引用变量，使用类中的引用数据成员，接受引用作为传递给函数和方法的参数，也可以从函数和方法返回引用。

#### 12.1.1 引用变量

引用变量必须在创建时就初始化，如下所示：

```
int x = 3;
int& xRef = x;
```

在上面这个赋值运算之后，xRef就是x的另一个名字了。使用xRef的任何代码使用的都是x的当前值。给xRef赋任何值都会改变x的值。比如，下面这段代码通过xRef把x设置为10。

```
xRef = 10;
```

如果在类的外部声明引用变量而不初始化，这是不允许的：

```
int& emptyRef; // DOES NOT COMPILE!
```

必须在分配引用时对其初始化。通常，引用是在声明时分配的，不过引用数据成员可以在包含该成

员的类的初始化列表中进行初始化。

除非引用指向一个 `const` 值，否则不能创建指向未命名值的引用，比如创建整数直接量（literal，也理解为字面量或立即数）的引用就无法通过编译：

```
int& unnamedRef = 5; // DOES NOT COMPILE
const int& unnamedRef = 5; // Works as expected
```

### 修改引用

引用总是指向初始化时指定的那个变量。一旦创建引用，就不能再修改。这个规则导致了一些令人困惑的语法。如果在声明引用时把一个变量“赋值”给了该引用，这个引用就会指向那个变量。但是，如果在这之后，又把一个变量赋值给了这个引用，该引用所指向的变量（译者注：还是原来的那个变量，而不是新变量）的值就会改变，即变成所赋的新变量的值。引用本身并不会改变，它不会更新为指向这个新变量。下面这段示例代码就说明了这个问题：

```
int x = 3, y = 4;
int& xRef = x;
xRef = y; // Changes value of x to 4. Doesn't make xRef refer to y.
```

你或许希望在赋值时取 `y` 的地址来绕过这条限制：

```
int x = 3, y = 4;
int& xRef = x;
xRef = &y; // DOES NOT COMPILE!
```

这段代码无法编译成功。`y` 的地址是指针，但是 `xRef` 声明为 `int` 变量的引用，而不是指针的引用。

有些程序员甚至想违反引用本身所固有的语义。如果把一个引用赋值给另一个引用会怎么样呢？是不是第一个引用会指向第二个引用指向的变量呢？你可能想试着编译下面这段代码：

```
int x = 3, z = 5;
int& xRef = x;
int& zRef = z;
zRef = xRef; // Assigns values, not references
```

最后一行代码不会改变 `zRef`。而是会把 `z` 的值设置为 3，因为 `xRef` 指向的是 `x`，它的值是 3。

引用指向的变量在初始化之后不能再改变，只能改变这个变量的值。

### 指针引用和引用指针

可以创建指向任何类型的引用，包括指针类型。下面是一个指向 `int` 指针的引用的例子：

```
int* intP;
int*& ptrRef = intP;
ptrRef = new int;
*ptrRef = 5;
```

这里使用的语法有点奇怪：你可能不太习惯看到 `*` 和 `&` 紧挨着出现。但是，其中的语义是一目了然的：`ptrRef` 是 `intP` 的引用，而 `intP` 是指向 `int` 的指针。修改 `ptrRef` 会改变 `intP`。指针引用很少用，但是有时候会很有用，这一点在本章的 12.1.3 节讨论。

注意，取引用的地址和取引用所指向变量的地址，这二者的结果是一样的。例如：

```
int x = 3;
int& xRef = x;
int* xPtr = &xRef; // Address of a reference is pointer to value
*xPtr = 100;
```

这段代码是取指向 x 的引用 (xRef) 的地址, 把 xPtr 设置为指向 x。把 100 赋值给 \* xPtr 会把 x 的值变为 100。

最后要注意, 不能声明指向引用的引用, 也不能声明引用指针 (指向引用的指针):

```
int x = 3;
int& xRef = x;
int&& xDoubleRef = xRef; // DOES NOT COMPILE!
int&* refPtr = &xRef; // DOES NOT COMPILE!
```

### 12.1.2 引用数据成员

在第 9 章曾经学到过, 类的数据成员可以是引用。但是如果引用不指向其他某个变量, 这样的引用是无法存在的。因此, 必须在构造函数初始化列表中初始化引用数据成员, 而不是在构造函数体中完成初始化。详细内容请参考第 9 章。

### 12.1.3 引用参数

C++ 程序员不常使用独立的引用变量或者引用数据成员。引用最通常的用法是作为函数和方法的参数。回忆一下, 默认的参数传递语义是传值 (pass-by-value): 函数接收到参数的副本。在修改这些参数时, 原来的实参保持不变。引用则允许你为传递给函数的实参指定另一种语义, 即传引用 (pass-by-reference)。使用引用参数时, 函数收到的是指向函数实参的引用。如果修改了这些引用, 其中的变化会反映到原始的实参变量上。例如, 下面这段代码是一个简单的交换函数, 其作用是交换两个 int 变量的值:

```
void swap(int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}
```

可以这样来调用该函数:

```
int x = 5, y = 6;
swap(x, y);
```

使用实参 x 和 y 调用函数 swap() 时, 参数 first 初始化为指向 x (x 的引用), 参数 second 初始化为指向 y (y 的引用)。函数 swap() 修改 first 和 second 时, 会真正改变 x 和 y。

我们知道, 不能使用常量来初始化常规的引用变量, 与此类似, 不能把常量作为实参传递给采用传引用语义的函数:

```
swap(3, 4); // DOES NOT COMPILE
```

#### 来自指针的引用

如果要把一个指针传递给函数或方法, 而该函数或方法需要的是一个引用, 此时就会出现一个很有普遍性的问题。在这种情况下, 可以简单地对指针解除引用, 这样就能把指针“转化”为引用。这样做

得到的是指针指向的值，随后编译器会用这个值来初始化引用参数。比如，可以如下调用函数 `swap()`：

```
int x = 5, y = 6;
int *xp = &x, *yp = &y;
swap(*xp, *yp);
```

### 传引用和传值

如果想要修改参数，并希望这些修改反映到函数或方法的实参变量上，此时就应该采用传引用。但是，不应该限制为只是在这种情况下才采用传引用。传引用可以避免复制函数实参，在某些情况下这会带来另外两个好处：

1. 效率。复制大的对象和结构（struct）时可能会花费很长时间。传引用只向函数传递指向对象或 struct 的指针，这会节省时间。

2. 正确性。不是所有的对象都允许传值。即使允许传值，也不见得就能正确地支持深复制。第 9 章曾介绍过，要支持深复制，有动态分配内存的对象必须提供定制的复制构造函数。

如果想充分发挥这两个优点，但是又不想修改原来的对象，可以在参数前面标以 `const`。这个主题会在本章的后续内容中详细介绍。

传引用的这些优点意味着，对于简单的内置类型（如 `int` 和 `double`），不需要修改实参，就应当使用传值。在其他所有的情况下都可以使用传引用。

### 12.1.4 引用返回类型

从函数或方法也可以返回引用。这样做的主要原因是出于效率考虑。不是返回一个完整的对象，而是从函数或方法返回对象的引用，这样就可以避免不必要的复制。当然，只有当前对象在函数结束之后仍然存在时才可以使用这个技术。

不要返回变量（比如在堆栈上自动分配的变量）的引用，因为函数结束时会撤销这个变量。

从函数或方法返回引用的第二个原因是想把它作为 lvalue（赋值语句的左值）直接赋值给返回值。

一些重载操作符通常都返回引用。在第 9 章中已经看到了这样一些例子，关于这个技术的更多应用请阅读第 16 章。

### 12.1.5 采用引用还是指针

C++ 中的引用大概是多余的，引用可以做的事情，几乎都能用指针完成。例如，可以这样来编写前面的 `swap()` 函数：

```
void swap(int* first, int* second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}
```

但是这段代码比起使用引用的 `swap()` 函数来要混乱得多，引用让程序清晰，易于理解。引用也比指针安全，不可能存在无效的引用，不需要明确地解除引用，所以不会遇到指针可能存在的解除引用错误。大部分情况下，可以使用引用而不是指针。与对象指针类似，对象引用甚至也支持多态性。惟一需要使用指针的情况是，需要改变指针指向的位置。回忆一下，你不能改变引用指向的变量。比如，在动态分配内存时，就需要在指针中存储指向结果的指针，而不是在引用中存储。



要看参数和返回类型中是采用指针合适还是采用引用合适，还有一种方法，就是考虑谁拥有 (own) 内存。如果收到变量的代码要负责释放与对象关联的内存，就必须接收对象的指针。如果收到变量的代码不必释放内存，则应该接收变量引用。

除非需要动态分配内存或者要在其他地方改变或释放指针指向的值，否则，都应使用引用而不是指针。

这条规则也适用于独立变量、函数或方法参数，以及函数或方法的返回值。

严格应用这条规则会带来一些我们不习惯的语法。考虑这样一个函数，它把一个 int 数组分为两个数组：一个偶数组成的数组和一个奇数组成的数组。这个函数并不知道源数组中有多少元素是奇数或者有多少元素是偶数，所以在检查完源数组之后，它应该动态分配目标数据的内存。该函数还应该返回两个新数组的大小。一共要返回四项数据：指向两个新数组的指针和两个新数组的大小。很明显，在这种情况下，必须采用传引用。使用规范的 C 语言方式来编写这个函数，应该如下所示：

```
void separateOddsAndEvens(const int arr[], int size, int** odds, int* numOdds,
                          int** evens, int* numEvens)
{
    int i;
    // First pass to determine array sizes
    *numOdds = *numEvens = 0;
    for (i = 0; i < size; i++) {
        if (arr[i] % 2 == 1) {
            (*numOdds)++;
        } else {
            (*numEvens)++;
        }
    }

    // Allocate two new arrays of the appropriate size.
    *odds = new int[*numOdds];
    *evens = new int[*numEvens];
    // Copy the odds and evens to the new arrays
    int oddsPos = 0, evensPos = 0;
    for (i = 0; i < size; i++) {
        if (arr[i] % 2 == 1) {
            (*odds)[oddsPos++] = arr[i];
        } else {
            (*evens)[evensPos++] = arr[i];
        }
    }
}
```

函数的最后四个参数就是“引用”参数。要改变它们指向的值，函数 separateOddsAndEvens() 必须解除对它们的引用，这会导致在函数体中出现一些让人很不喜欢的语法。

此外，想要调用函数 separateOddsAndEvens() 时，必须向函数传递两个指针的地址，这样函数才能改变具体的指针，此外还需要传递两个 int 的地址，以便函数改变具体的 int。

```
int unSplit[10] = {1, 2, 3, 4, 5, 6, 6, 8, 9, 10};
int *oddNums, *evenNums;
int numOdds, numEvens;

separateOddsAndEvens(unSplit, 10, &oddNums, &numOdds, &evenNums, &numEvens);
```



如果你对这样的语法很不满意（肯定会有不满的），可以使用引用编写一个同样的函数，来获得真正的传引用语义：

```
void separateOddsAndEvens(const int arr[], int size, int& odds, int& numOdds,
    int& evens, int& numEvens)
{
    int i;
    numOdds = numEvens = 0;
    for (i = 0; i < size; i++) {
        if (arr[i] % 2 == 1) {
            numOdds++;
        } else {
            numEvens++;
        }
    }

    odds = new int[numOdds];
    evens = new int[numEvens];

    int oddsPos = 0, evensPos = 0;
    for (i = 0; i < size; i++) {
        if (arr[i] % 2 == 1) {
            odds[oddsPos++] = arr[i];
        } else {
            evens[evensPos++] = arr[i];
        }
    }
}
```

在这种情况下，参数 odds 和 evens 都是 int\* 的引用。separateOddsAndEvens() 不用显式地解除引用，就可以修改作为函数实参的 int\*（通过引用）。同样的逻辑还可以应用于 numOdds 和 numEvens，它们都是 int 的引用。

有了采用引用的 separateOddsAndEvens() 函数，就不需要传递指针或 int 的地址了。引用参数会自动地处理：

```
int unSplit[10] = {1, 2, 3, 4, 5, 6, 6, 8, 9, 10};
int *oddNums, *evenNums;
int numOdds, numEvens;
```

```
separateOddsAndEvens(unSplit, 10, oddNums, numOdds, evenNums, numEvens);
```

## 12.2 关键字疑点

C++ 中有两个关键字看起来比其他的关键字更容易带来困惑：const 和 static。这两个关键字都有几个不同的含义，每种用法都有一些微妙的地方需要很好地理解，这很重要。

### 12.2.1 关键字 const

关键字 const 是“constant”的简写，它指定或者要求用其声明的内容保持不变。就像在这本书的许多地方看到的那样（在实际的代码中可能也是这样），关键字 const 有两种不同但相关的用法：用于标识变量和用于标识方法。本节将明确讨论这两种含义。

### const 变量

可以使用 `const` 来声明不能对变量进行修改，以此来“保护”变量。就像在第 1 章和第 7 章介绍的，关键字 `const` 的一个重要用法就是代替 `#define` 来声明常量。`const` 的这个用法正是最直接的应用。比如，可以这样来声明常量 `PI`：

```
const double PI = 3.14159;
```

可以把任何变量标识为 `const`，包括全局变量和类的数据成员。

也可以使用 `const` 来指定函数或方法的参数应该保持不变。在第 1 章和第 9 章以及这本书的其他地方应该已经看到过这种应用的例子。

### const 指针

变量通过指针包含一层或多层间接引用时，应用 `const` 就变得有点复杂了。看一下下面这几行代码：

```
int* ip;  
ip = new int[10];  
ip[4] = 5;
```

假设你决定对 `ip` 应用 `const`（即把 `ip` 声明为 `const`）。你可能会怀疑这样做是否有用。暂时把这种怀疑放在一边，先来考虑它到底有什么含义。你是不是不想让变量 `ip` 改变，或者不希望 `ip` 指向的值有变化？也就是说，是不是想阻止前一个例子的第二行或者第三行代码执行？

为了阻止修改 `ip` 指向的值（像第三行代码那样），可以给 `ip` 的声明增加关键字 `const`：

```
const int* ip;  
ip = new int[10];  
ip[4] = 5; // DOES NOT COMPILE!
```

现在就不能改变 `ip` 指向的值了。

还有一种方法：

```
int const* ip;  
ip = new int[10];  
ip[4] = 5; // DOES NOT COMPILE!
```

把 `const` 放在 `int` 之前还是之后对其功能来说没有什么区别。

如果想把 `ip` 自己（不是它指向的值）声明为 `const`，需要这样编写代码：

```
int* const ip = NULL;  
ip = new int[10]; // DOES NOT COMPILE!  
ip[4] = 5;
```

既然不能改变 `ip` 本身，编译器就要求在声明这个变量时要对其初始化。

也可以像下面这样，把指针及其指向的值都声明为 `const`：

```
int const* const ip = NULL;
```

另外一种语法如下：

```
const int* const ip = NULL;
```

虽然这样写看起来可能有点难以理解，其实有一条非常简单的规则：关键字 `const` 应用于紧挨着

const 的左侧第一项。再看一遍这行代码：

```
int const* const ip = NULL;
```

从左到右来看一下，第一个 const 紧挨着放在单词 int 的右侧。因此，它应用于 ip 指向的 int。所以，这个 const 声明，不能改变 ip 指向的值。第二个 const 位于符号\* 的右侧。因此，它应用于指向 int 的指针，也就是变量 ip。所以，第二个 const 指定，不能修改 ip 自身（即指针）。

Const 应用于紧挨着位于其左侧的间接层（间接引用）。

这条规则之所以让人迷惑，是因为存在一个例外：第一个 const 可以位于变量前面，就像下面这样：

```
const int* const ip = NULL;
```

这个“例外的”语法比起其他的语法来说用的要多得多。

可以把这条规则扩展为任意多个间接层。例如：

```
const int * const * const * const ip = NULL;
```

### const 引用

应用于引用的 const 关键字通常比应用于指针 const 关键字要简单，这主要有两个原因：第一，引用默认就是 const 的，也就是说不能改变它们指示的变量。所以，C++ 不允许显式地用 const 来标识引用变量。第二，引用一般只有一个间接层。前面已经解释过，不能创建对引用的引用。要得到多重间接层（间接引用），惟一的办法就是创建指针的引用。

因此，C++ 程序员谈到“const 引用”时，他们的意思可能是：

```
int z;  
const int& zRef = z;  
zRef = 4; // DOES NOT COMPILE
```

通过对 int 应用 const，可以防止对 zRef 赋值，就像上面的代码那样。记住，const int& zRef 与 int const& zRef 是等价的。但是要注意，用 const 标识 zRef 对 z 没有影响。仍然可以直接改变 z 来修改 z 的值，而不是通过引用。

const 引用最通常的用法就是作为参数，在这种情况下 const 引用的用处很大。如果为了提高效率，想要按引用来传递某个变量，但是又不希望修改这个变量，就可以把它设置为 const 引用。例如：

```
void doSomething(const BigClass& arg)  
{  
    // Implementation here  
}
```

把对象作为参数传递时，默认的做法应该是把传递 const 引用。只有确实需要改变对象时才应该去掉 const。

### const 方法

就像在第 9 章介绍的一样，可以用 const 来标识类方法。这样来声明方法可以防止方法修改类中不可变（non-mutable）的数据成员。具体的例子请参考第 9 章。

### 12.2.2 关键字 static

尽管在 C++ 中关键字 `const` 有几种用法，不过如果认为 `const` 的含义就是“不变的”，那么所有的用法都是相关而且有意义的。关键字 `static` 则不同，在 C++ 中它有三种用法，而且看起来都是不相关的。

#### static 数据成员和方法

就像在第 9 章读到的，可以为类声明静态数据成员和静态方法。静态数据成员不像非静态的数据成员，它们不是每一个对象的组成部分。相反，这个数据成员只有一个副本而已，它存在于该类的所有对象之外。

`static` 方法也是类似的，这些方法是类层次上的，而不是对象层次上的。静态方法不会在具体对象的上下文中执行。

第 9 章中提供了一些静态成员和静态方法的例子。

#### static 连接

在了解对连接使用关键字 `static` 之前，需要先了解 C++ 中连接 (linkage) 的概念。在第 1 章已经学到，C++ 的每个源文件都是独立编译的，得到的对象文件要连接在一起。C++ 源文件中的每个名字，包括函数和全局变量，都有一个连接，可能是内部 (internal) 连接，也可能是外部 (external) 连接。外部连接是指，对于其他源文件，这个名字是可用的。内部连接 (也称为静态连接 (static linkage)) 是指，对于其他源文件，这个名字不可用。函数和全局变量默认都有外部连接。但是，可以在声明前面加上关键字 `static`，来指定内部 (或静态) 连接。比如，假设有两个源文件：FirstFile.cpp 和 AnotherFile.cpp。下面是源文件 FirstFile.cpp 的代码：

```
// FirstFile.cpp

void f();

int main(int argc, char** argv)
{
    f();
    return (0);
}
```

注意，这个文件给出了方法 `f()` 的原型，但是没有给出其定义。

下面是源文件 AnotherFile.cpp 的代码：

```
// AnotherFile.cpp

#include <iostream>
using namespace std;

void f();

void f()
{
    cout << "f\n";
}
```

这个文件不仅给出了方法 `f()` 的原型，还提供了其定义。需要说明，在两个不同的文件中编写同一个方法的原型是合法的。如果每个源文件中都用“`#include`”包含了一个头文件，并把方法的原型放在这个头文件中，预处理器所做的正是这个工作，其作用就是在不同的源文件中有同一个方法的原型。使用

头文件的原因是维护原型的副本（并保持同步）更为容易。不过，对于这个例子，我们没有使用头文件。

这些源文件都能无错误地通过编译，程序连接也没什么问题，因为方法 `f()` 有外部连接，`main()` 函数可以从不同的文件调用它。

然而，假设在源文件 `AnotherFile.cpp` 中对方法 `f()` 使用了关键字 `static`：

```
// AnotherFile.cpp
#include <iostream>
using namespace std;
```

```
static void f();
```

```
void f()
{
    cout << "f\n";
}
```

现在，尽管编译每个源文件时都没有编译错误，但是连接步骤不能成功，因为方法 `f()` 有内部（`static`）连接，这样源文件 `FirstFile.cpp` 中就不能使用这个方法。定义了 `static` 方法但是在源文件中没有使用时，有些编译器会发出警告（指出这些方法不应该用 `static` 声明，因为可能会在其他地方使用这些方法）。

注意，不需要在 `f()` 的定义前面重复关键字 `static`。只要函数名第一次出现时前面有 `static`，以后这个函数名前面就不需要重复这个关键字了。

现在已经学习了关于使用 `static` 的所有知识，C++ 委员会最终意识到，关键字 `static` 的含义过多，所以不建议使用 `static` 的这个特别用法，听到这个消息你会很高兴。这意味着，现在关键字 `static` 仍然是 C++ 标准的组成部分，但是不能保证将来也是这样。然而，许多遗留的 C++ 代码仍然在采用这种方式使用关键字 `static`。

另一种方法是采用匿名命名空间（`anonymous namespace`）来达到相同的效果。即把变量和函数包装在一个未命名的命名空间中，而不是用关键字 `static` 标识这些变量和函数，就像下面这样：

```
// AnotherFile.cpp
#include <iostream>
using namespace std;
```

```
namespace {
    void f();
```

```
    void f()
    {
        cout << "f\n";
    }
}
```

声明了匿名命名空间中的实体之后，可以在同一个源文件中的任意一个地方访问这些实体，但是在其他源文件中不能访问。这些语义与通过关键字 `static` 得到的语义是相同的。

### 关键字 `extern`

还有一个与 `static` 相关的关键字：`extern`。它看起来应该是与 `static` 相对立的，`extern` 用来为位于它前面的名字声明外部连接。有些情况下就可以这样来使用 `extern`。比如，`const` 和 `typedef` 默认的都有内部连接，所以可以使用 `extern` 为其指定外部连接。

但是 `extern` 有些复杂。把一个名字指定为 `extern` 时，编译器会把它当作声明而不是定义来对待。对于变量来说，这意味着编译器不会给变量分配空间。必须为变量提供没有关键字 `extern` 的另外的定义。



例如：

```
// AnotherFile.cpp
extern int x;
int x = 3;
```

也可以在包括有 extern 的这行代码中初始化 x，这样它可以同时作为声明和定义：

```
// AnotherFile.cpp
extern int x = 3;
```

这个文件中的 extern 并不是十分有用，因为不管怎样默认地 x 都有外部连接。需要在其他的源文件中使用 x 时，才会体现出 extern 的真正用处：

```
// FirstFile.cpp
#include <iostream>
using namespace std;

extern int x;

int main(int argc, char** argv)
{
    cout << x << endl;
}
```

源文件 FirstFile.cpp 使用了 extern 声明，所以它可以使用 x。为了能在 main() 函数中使用 x，编译器需要 x 的一个声明。但是，如果没有用关键字 extern 声明 x，编译器就会认为这是一个定义，会给 x 分配内存空间，这就会导致连接失败（因为现在在全局作用域内有两个 x 变量）。有了 extern，就使得变量在多个源文件中都可以进行全局访问了。

不过，我们建议，尽可能地不要使用全局变量。全局变量容易使人迷惑，也很容易出错，尤其是在大型程序中。要完成这样一些功能，应该使用 static 类成员和方法。

### 函数中的 static 变量

在 C++ 中，关键字 static 的最后一种用法是创建局部变量，只在进入和退出变量作用域之间维护变量的值。函数内部的静态变量就像是只能从该函数访问的全局变量一样。静态变量的一种通常用法是“记住”是否已经为一个函数完成过特定的初始化。比如，采用这种技术的代码可能就像下面这样：

```
void performTask()
{
    static bool initied = false;

    if (!initied) {
        cout << "initing\n";
        // Perform initialization.
        initied = true;
    }

    // Perform the desired task.
}
```

然而，static 变量往往让人很糊涂，通常还有更好的方法来建立代码，而避免使用 static 变量。在这种情况下，可能想在编写类时，编写一些构造函数来完成所需的初始化工作。



要避免使用独立的 static 变量。应在对象内维护变量状态。

### 12.2.3 非局部变量的初始化顺序

在结束对 static 数据成员和全局变量的讨论之前，我们来看一下这些变量的初始化顺序。程序中所有的全局变量和 static 类数据成员都是在 main() 函数开始运行之前初始化的。给定源文件中的变量按照它们在该文件中出现的顺序初始化。比如，在下面这个文件 Demo 中，会保证：x 在 y 之前初始化。

```
// source1.cpp  
  
class Demo  
{  
public:  
    static int x;  
};  
int Demo::x = 3;  
int y = 4;
```

然而，C++ 并没有指定也不能保证不同源文件中非局部变量的初始化顺序。如果在一个源文件中有一个全局变量 x，在另一个源文件中有一个全局变量 y，你将无从知道哪一个变量首先初始化。通常，没有指定这种顺序不会引起人们的注意。但是如果一个全局变量或静态变量依赖于另一个变量，就可能会有问题。回忆一下，对象的初始化意味着运行对象的构造函数。全局对象的构造函数可能会访问另一个全局对象，认为后者已经构造。如果这两个全局对象是在两个不同的源文件中声明的，就不能认为一个对象会在另一个对象之前已经构造。

不同源文件中非局部变量的初始化顺序并不确定。

## 12.3 类型与类型强制转换

第 1 章回顾了 C++ 的基本类型。第 8 章介绍了如何利用类编写自己的类型。本节我们来研究类型的两个更复杂的方面：typedef 与类型强制转换。

### 12.3.1 typedef

typedef 为既有的类型提供了一个新的名字，可以把 typedef 简单地认为是给现有类型名引入一个同义词的语法。typedef 没有创建新类型——它只是提供了引用原类型的新方法。新的类型名和原来的类型名可以交替使用。使用新类型名创建的变量与使用原类型名创建的变量是完全兼容的。

你可能会感到奇怪，为什么前一段对 typedef 的定义如此简单。你可能在自己的代码中已经使用过 typedef，或者至少见过使用 typedef 的代码，它们看起来并不是那么简单啊！然而，如果仔细分析一下所有的用法就会发现，他们只是提供了一个备用的类型名。

类型名最常见的用法是当实际的类型名很麻烦时可以提供一個可管理的名字。这种情况通常会在模板中发生。例如，假设需要使用第 11 章的 Grid 模板创建一个电子表格，也就是 SpreadsheetCell 类型的 Grid。没有 typedef 的话，如果要引用这个 Grid 类型来声明变量、指定函数参数以及其他类似的情况，就必须写作为 Grid<SpreadsheetCell>；

```
int main(int argc, char** argv)
```

```
{  
    Grid<SpreadsheetCell> mySpreadsheet;  
    // Rest of the program . . .  
}
```

```
void processSpreadsheet(const Grid<SpreadsheetCell>& spreadsheet)
```

```
{  
    // Body omitted  
}
```

使用 typedef 可以建立更短，但更有意义的名字：

```
typedef Grid<SpreadsheetCell> Spreadsheet;
```

```
int main(int argc, char** argv)
```

```
{  
    Spreadsheet mySpreadsheet;  
    // Rest of the program . . .  
}
```

```
void processSpreadsheet(const Spreadsheet& spreadsheet)
```

```
{  
    // Body omitted  
}
```

typedef 的一个巧妙之处就是，类型名可以包含作用域限定符。比如，在第 9 章中，曾经这样用过 typedef：

```
typedef Spreadsheet::SpreadsheetCell SCell;
```

这个 typedef 关键字创建了一个短名字 SCell，用来在 Spreadsheet 的作用域内引用 SpreadsheetCell 类型。

STL 大量使用了 typedef 来定义较短的类型名称。例如，string 就是用 typedef 定义的，如下所示：

```
typedef basic_string<char> string;
```

### 12.3.2 类型强制转换

第 1 章已经解释过，C 中使用()进行类型强制转换，这种老式的转换方法在 C++ 中仍然是可用的。但是，C++ 还提供了四种新的类型强制转换方法：static\_cast、dynamic\_cast、const\_cast 和 reinterpret\_cast。你应该使用 C++ 风格的类型强制转换而不是 C 风格的类型强制转换，因为 C++ 的类型强制转换会完成更多的类型检查，这样可以得到更好的代码。

这一节将介绍每种类型强制转换的用途，并说明各种类型强制转换在什么情况下使用。

const\_cast

const\_cast 是最直接的类型强制转换。使用 const\_cast 可以去除变量的常量性 (const-ness)。这是四种类型强制转换中惟一一种允许去除变量常量性的类型强制转换。当然，从理论上讲，应该不会需要进行 const 类型强制转换。如果变量已经用 const 声明，就应该是保持不变的。但是在实践中，有时候你会发现自己处于这样一种情况：函数指定为要取一个 const 变量，但是接着这个 const 变量必须传递给一个取非 const 变量的函数。“正确”的解决办法是在程序中保持 const 的一致性，但是不能总是这样做，尤其是使用第三方的库时更是这样。因此，有时候需要去除变量的常量性。下面是一个例子：

```

void g(char* str)
{
    // Function body omitted for brevity
}
void f(const char* str)
{
    // Function body omitted for brevity
    g(const_cast<char*>(str));
    // Function body omitted for brevity
}

```

### static\_cast

可以使用 `static_cast` 来显式地完成 C++ 语言直接支持的转换。比如，如果编写了一个算术表达式，在这个表达式中需要把 `int` 转换为 `double` 来避免整数除法，可以使用 `static_cast`：

```

int i = 3;
double result = static_cast<double>(i) / 10;

```

也可以使用 `static_cast` 来显式地完成用户定义构造函数或者转换例程所允许的转换。比如，若类 A 有一个构造函数，这个构造函数取类 B 的一个对象，那么可以使用 `static_cast` 把 B 对象转换为一个 A 对象。然而，在需要进行这种转换的大部分情况下，编译器都会自动完成转换。

`static_cast` 的另一种用法是在继承层次结构中完成向下类型强制转换。例如：

```

class Base
{
public:
    Base() {}
    virtual ~Base() {}
};

class Derived : public Base
{
public:
    Derived() {}
    virtual ~Derived() {}
};

int main(int argc, char** argv)
{
    Base* b;
    Derived* d = new Derived();

    b = d; // Don't need a cast to go up the inheritance hierarchy
    d = static_cast<Derived*>(b); // Need a cast to go down the hierarchy

    Base base;
    Derived derived;

    Base& br = base;
    Derived& dr = static_cast<Derived&>(br);

    return (0);
}

```

这些类型强制转换可以应用于指针，也可以应用于引用。但是不能处理对象本身。

注意，使用 `static_cast` 的这些类型强制转换不会完成运行时类型检查。这些类型强制转换允许把任何基（Base）指针转换为派生（Derived）指针，也可以把基（Base）引用转换为派生（Derived）引用，甚至在运行时基指针或引用实际上并不是派生指针或引用时也允许这样转换。要想安全地完成类型强制转换，同时进行运行时类型检查，请使用 `dynamic_cast`。

`static_cast` 不是全能的。不能使用 `static_cast` 把一种类型的指针转换为另一种无关的类型。不能使用 `static_cast` 把指针转换为指向 `int`。不能使用 `static_cast` 直接把一种类型的对象转换为另一种类型的对象。不能使用 `static_cast` 把一个 `const` 类型强制转换为非 `const` 类型。基本说来，根据 C++ 的类型规则，任何没有意义的转换 `static_cast` 都做不到。

#### `reinterpret_cast`

`reinterpret_cast` 功能比 `static_cast` 更强，随之而来的是安全性较低。可以使用 `reinterpret_cast` 来完成满足以下条件的类型强制转换：即按照 C++ 类型规则来说，从理论上讲这种转换是不允许的，但是在某种场合下它们对于程序员可能很有意义。例如，可以把一种指针类型强制转换为另外某种指针类型，即使在继承层次结构中它们不相关也可以。类似地，可以把一种类型的引用转换为另一种类型的引用，即使这两种引用类型不相关也是允许的。还可以把指针转换为 `int`，或者是把 `int` 转换为指针。下面是几个例子。

```
class X {};  
class Y {};  
  
int main(int argc, char** argv)  
{  
    int i = 3;  
  
    X x;  
    Y y;  
  
    X* xp;  
    Y* yp;  
  
    // Need reinterpret cast to perform pointer conversion from unrelated classes  
    // static_cast doesn't work.  
    xp = reinterpret_cast<X*>(yp);  
  
    // Need reinterpret_cast to go from pointer to int and from int to pointer  
    i = reinterpret_cast<int>(xp);  
    xp = reinterpret_cast<X*>(i);  
  
    // Need reinterpret cast to perform reference conversion from unrelated classes  
    // static_cast doesn't work.  
    X& xr = x;  
    Y& yr = reinterpret_cast<Y&>(x);  
  
    return (0);  
}
```

在使用 `reinterpret_cast` 时要格外小心，因为它会把原始的位（bit）重新解释为不同的类型，而不完成任何类型检查。

#### `dynamic_cast`

就像讨论 `static_cast` 时提到的，使用 `dynamic_cast` 进行类型强制转换时，会在继承层次结构中对类型

强制转换完成运行时类型检查。可以使用 `dynamic_cast` 来对指针或引用进行类型强制转换。`dynamic_cast` 会在运行时检查底层对象的运行时类型信息。如果类型强制转换没有意义，`dynamic_cast` 会返回 `NULL`（对于指针转换）或者抛出 `bad_cast` 异常（对于引用转换）。

注意，运行时类型信息是存储在对象的 `vtable` 里面的。因此，要使用 `dynamic_cast`，类必须至少有一个虚（`virtual`）函数。

下面来看几个例子。

```
#include <typeinfo>
#include <iostream>
using namespace std;

class Base
{
public:
    Base() {};
    virtual ~Base() {}
};

class Derived : public Base
{
public:
    Derived() {}
    virtual ~Derived() {}
};

int main(int argc, char** argv)
{
    Base* b;
    Derived* d = new Derived();

    b = d;
    d = dynamic_cast<Derived*>(b);

    Base base;
    Derived derived;

    Base& br = base;

    try {
        Derived& dr = dynamic_cast<Derived&>(br);
    } catch (bad_cast&) {
        cout << "Bad cast!\n";
    }

    return (0);
}
```

在上面这个例子中，第一个转换应该能够成功完成，而第二个会抛出一个异常。第 15 章会详细讨论异常处理。

需要注意的是，可以使用 `static_cast` 或 `reinterpret_cast` 沿着继承层次结构完成同样的向下类型强制转换。它们与 `dynamic_cast` 的区别是，`dynamic_cast` 要完成运行时（动态）类型检查。



## 类型强制转换小结

表 12-1 总结了在不同情况下应该使用哪一种类型强制转换。

表 12-1

类型强制转换情况	转换方法
删除常量性	const_cast
语言支持的显式类型强制转换（比如 int 转换为 double，int 转换为 bool）	static_cast
用户定义的构造函数或转换所支持的显式类型强制转换	static_cast
一个类的对象转换为另一个（不相关的）类的对象	无法完成
相同的继承层次结构中，一个类的对象指针转换为另一个类的对象指针	static_cast 或 dynamic_cast
相同的继承层次结构中，一个类的对象引用转换为另一个类的对象引用	static_cast 或 dynamic_cast
一种类型的指针转换为不相关的另一种类型的指针	reinterpret_cast
一种类型的引用转换为不相关的另一种类型的引用	reinterpret_cast
指针转换为 int/int 转换为指针	reinterpret_cast
函数指针的转换	reinterpret_cast

## 12.4 解析作用域

作为一名 C++ 程序员，需要熟悉作用域（scope）的概念。程序中的每个名字，包括变量、函数和类名，都是在某个作用域之内的。作用域是根据命名空间、函数定义和类定义来创建的。访问变量、函数或者类时，首先在最内层作用域内查找要访问的名字，然后再在较外层作用域中查找，以此类推，直到全局作用域（global scope）。不在任何命名空间、函数或者类中的名字都在全局作用域内。

有时候，一些作用域中的名字会隐藏其他作用域中同样的名字（译者注：例如，一个作用域中的变量会隐藏其他作用域中的同名变量）。还有一些情况是，对于程序中某一行特定的代码，需要的作用域并非这行代码的默认作用域。如果不想让一个名字使用默认的作用域解析，可以使用作用域解析操作符::，为这个名字限定一个特定的作用域。比如，要访问类的静态方法，可以用类名（其作用域）和作用域解析操作符给方法名加上一个前缀。

```
class Demo
{
public:
    static void method() {}
};

int main(int argc, char** argv)
{
    Demo::method();

    return (0);
}
```

在本书中，还有其他一些作用域解析的例子。但是有一点需要特别注意，这就是访问全局作用域。全局作用域是未命名的，所以没有专门用来访问作用域的方法。不过可以单独使用作用域解析操作符



(没有名字前缀), 这样就总是引用全局作用域。举例如下:

```
int name = 3;

int main(int argc, char** argv)
{
    int name = 4;

    cout << name << endl; // Accesses local name
    cout << ::name << endl; // Accesses global name

    return (0);
}
```

## 12.5 头文件

头文件是为子系统或者代码片断提供抽象接口的一种机制。使用头文件一大难点是, 要避免同一个头文件的循环引用和多重包含。比如, 你要负责编写 `Logger` 类, 该类的任务是记录所有的错误日志信息。最后 `Logger` 类可能要使用另一个类 `Preferences`, 由这个类负责跟踪用户设置。而类 `Preferences` 可能又会通过另一个头文件间接使用类 `Logger`。

如下面这段代码所示, `#ifndef` 机制可以用于避免循环包含和多重包含。在每个头文件的一开始, `#ifndef` 指令就会进行检查, 看某个特定的键是否未定义。如果所检查的键已经定义, 编译器会跳到与之匹配的 `#endif`, `#endif` 一般放在文件的结尾。如果所检查的键没有定义, 该文件则会定义这个键, 这样下一次包含这个文件时就会直接跳过 (因为已经定义了相应的键)。

// Logger.h

```
#ifndef __LOGGER__
#define __LOGGER__

#include "Preferences.h"

class Logger
{
public:
    static void setPreferences(const Preferences& inPrefs);
    static void logError(const char* inError);
};

#endif // __LOGGER__
```

要避免头文件的这些问题, 另一种工具是超前引用 (forward reference)。如果需要引用一个类, 但是不能包含它的头文件 (比如, 因为这个类相当依赖于你正在编写的类), 那么可以只是告诉编译器存在这样的一个类, 而不是通过 `#include` 机制提供正式的定义。当然, 还不能在代码中真正使用这个类, 因为编译器可以说对这个类一无所知, 只是知道所有代码都已连接之后会有这样一个类存在。不过, 在类定义中还是可以使用指向这个类的指针或者引用。在下面这段代码中, 类 `Logger` 引用了类 `Preferences`, 但是没有包含它的头文件。

// Logger.h

```
#ifndef __LOGGER__
```

```
#define __LOGGER__

class Preferences;

class Logger
{
public:
    static void setPreferences(const Preferences& inPrefs);
    static void logError(const char* inError);
};

#endif // __LOGGER__
```

## 12.6 C 实用工具

回忆一下，C++ 是 C 的超集，因此 C++ 包含了 C 的所有功能。但是 C 的几个隐含特性在 C++ 中却没有对应的“替代品”，而这几个特性有时候是很有用的。本节来分析一下其中的两个特性：变长参数列表和预处理器宏。

### 12.6.1 变量长度参数列表

看一下 `<stdio.h>` 中的 C 函数 `printf()`。可以基于任意多个参数来调用该函数：

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("int %d\n", 5);
    printf("String %s and int %d\n", "hello", 5);
    printf("Many ints: %d, %d, %d, %d, %d\n", 1, 2, 3, 4, 5);
}
```

C++ 提供了有关的语法和一些实用工具宏来编写参数个数可变的函数。这些函数通常看起来很像 `printf()` 函数。虽然不应该频繁使用这个特性，但是有些情况下使用这个特性会非常有用。例如，假设需要快速地编写一个调试函数，但是不要求十分完善，它只需完成如下功能：如果设置了调试标志，该函数打印字符串到 `stderr`，如果没有设置调试标志，则不做任何操作。这个函数应该能够根据个数和类型都任意的参数来打印字符串。一个简单的实现如下：

```
#include <stdio.h>
#include <stdarg.h>

bool debug = false;

void debugOut(char* str, ...)
{
    va_list ap;
    if (debug) {
        va_start(ap, str);
        vfprintf(stderr, str, ap);
        va_end(ap);
    }
}
```

首先要注意，`debugOut()`的原型包含了一个类型和名字都已经指定的参数 `str`，后面是...（省略号）。它们表示任意数量和类型的参数。要访问这些参数，必须使用在 `<stdarg.h>` 中定义的宏。可以声明 `va_list` 类型的变量，并通过调用 `va_start()` 来初始化该变量。`va_start()` 的第二个参数必须是参数列表中最右边的命名变量。所有函数都至少需要一个命名参数。函数 `debugOut()` 只是把这个列表传递给 `vfprintf()`（`<stdio.h>` 中的一个标准函数）。在这个函数结束之后，它调用 `va_end()` 来结束对变长参数列表的访问。在调用 `va_start()` 之后必须调用 `va_end()` 来确保函数调用栈最后保持一致状态。

可以像下面这样使用该函数：

```
int main(int argc, char** argv)
{
    debug = true;
    debugOut("int %d\n", 5);
    debugOut("String %s and int %d\n", "hello", 5);
    debugOut("Many ints: %d, %d, %d, %d, %d\n", 1, 2, 3, 4, 5);

    return (0);
}
```

### 访问参数

如果要自己访问实参，可以使用 `va_arg()`。比如有一个函数，它取任意数量的 `int` 参数，并把它们打印出来：

```
#include <iostream>
using namespace std;

void printInts(int num, ...)
{
    int temp;
    va_list ap;
    va_start(ap, num);
    for (int i = 0; i < num; i++) {
        temp = va_arg(ap, int);
        cout << temp << " ";
    }
    va_end(ap);
    cout << endl;
}
```

可以如下调用 `printInts()`：

```
printInts(5, 5, 4, 3, 2, 1);
```

### 为什么不要使用变长参数列表

访问变长参数列表并不是很安全。从函数 `printInts()` 可以看出来，存在以下几个风险：

- 你不知道参数的个数。对于 `printInts()`，必须相信调用者在第一个参数中传递的参数个数是正确的。对于 `debugOut()`，如果字符数组中有格式化代码，必须相信调用者传递的参数个数与字符数组中格式化代码指定的参数个数相符。
- 你不知道参数的类型。`va_arg()` 会取一个类型，用来解释当前遇到的值。不过，可以告诉 `va_arg()` 把这个值解释为任何类型。并没有什么方法可以验证类型是否正确。

要避免使用变长参数列表。更可取的方法是传入一个变量数组或向量。

### 12.6.2 预处理宏

可以使用C++的预处理器来编写宏 (macro)，宏有点类似于函数。以下是一个例子。

```
#define SQUARE(x) ((x) * (x)) // No semicolon after the macro definition!

int main(int argc, char** argv)
{
    cout << SQUARE(4) << endl;

    return (0);
}
```

宏是从C保留下来的内容，它与inline函数十分类似，只是不进行类型检查而已，预处理器会简单地把宏调用代之以宏的展开形式。预处理器并没有应用真正的函数调用语义。这个行为会导致不可预料的结果。比如，如果基于2+2而不是4来调用宏SQUARE会怎么样呢，就像下面这样：

```
cout << SQUARE(2 + 2) << endl;
```

你可能认为SQUARE的计算结果为16，的确是这样。但是，如果在宏定义中漏掉了小括号会怎么样呢？就像下面这样：

```
#define SQUARE(x) (x * x)
```

现在调用SQUARE(2+2)会得出8，而不是16。要记住，宏只是简单地原样展开，根本没有考虑函数调用语义。这意味着，宏体中的任何x都被2+2所代替，这会导致以下的展开结果：

```
cout << 2 + 2 * 2 + 2 << endl;
```

根据正确的操作顺序，这行代码会先完成乘法运算，然后是加法，最后的结果就是8，而不是16。

宏还会带来调试问题，因为你编写的代码并不是编译器看到的代码，或者不是调试器见到的代码（因为存在着预处理器的搜索和替换行为）。出于这些原因，应该避免完全使用宏来代替内联函数。之所以在这里介绍宏的有关细节，原因是确实有许多代码都采用了宏。要阅读和维护这些代码就需要理解宏。

## 12.7 小结

本章解释了C++中最让人困惑的几个方面。你在本章中学到了更多的C++语法细节。其中的一些内容，比如引用的有关细节、const、作用域解析、C++风格类型强制转换的内容、头文件的有关技术等会在程序中经常使用。其他一些信息，比如static和extern的用法，如何编写变长参数列表，如何编写预处理宏等等，理解这些内容很重要，但是不要在日常的程序中频繁使用。不论怎样，既然已经懂得了这些技术细节，接下来就可以泰然自若地应对本书后面章节所介绍的C++高级技术了。



# 第三部分

## 掌握 C++ 高级特性

### 第 13 章 有效的内存管理

在许多情况下，用 C++ 编程就像在没有道路的情况下开车一样。的确，你可以去往想去的任何地方，但是没有行车线或者红绿灯的保护，你可能会伤害到自己。C++ 像 C 一样，为程序员提供了一种像传球一样的方法。C++ 语言认为你知道自己正在干什么。即使你编写的程序可能会产生问题，C++ 也是允许的，因为 C++ 有非凡的灵活性，而且通过牺牲安全性来换取性能。

内存分配与管理是 C++ 程序设计中特别容易产生错误的方面。要编写高质量的 C++ 程序，专业的 C++ 程序员需要懂得内存存在后台如何工作。这一章研究了内存管理中的输入输出。你将学到动态内存的陷阱，以及可以采用哪些技术来避免和消除这样一些问题。

本章首先对使用和管理内存的不同方法提供了一个概述。接下来将介绍数组与指针之间常常让人很迷惑的关系。随后解释如何创建和管理 C 风格的字符串。接着是从底层介绍内存如何操作。最后一节介绍了在内存管理中可能会遇到的一些特定的问题，并提出了许多解决方法。

#### 13.1 使用动态内存

在学习程序设计时，动态内存经常是缺乏经验的程序员所面对的主要障碍。内存是计算机的低级组件，遗憾的是，即使在像 C++ 这样的高级程序设计语言中，内存也会“不太合适”地探出头来。许多程序员只是懂得一些动态内存的知识就觉得足够了。他们要么不敢触及使用动态内存的数据结构，要么只好借助于反复的尝试、出错、修正错误后再尝试的方法来完成程序。

在程序中使用动态内存有两个主要的优点：

- 动态内存可以在不同的对象与函数之间共享。
- 动态分配的内存空间的大小可以在运行时确定。

如果能很好地理解 C++ 中动态内存是如何工作的，这对于成为一名专业 C++ 程序员可谓是一个基本条件。

##### 13.1.1 如何描述内存

对象在内存中是个什么样子？如果能够在脑海中对此有一个模型，理解动态内存就容易得多了。在本书中，内存单元表示为一个带有标号的方框。这个标号指示了对应该内存（单元）的变量名。方框内的数据显示了该内存（单元）的当前值。



比如，图 13-1 给出了在执行下面这行代码之后的内存状态：

```
int i = 7;
```

回忆一下在第 1 章讲到的，变量 *i* 是在栈（stack）中分配的，因为它声明为简单类型，而不是用关键字 *new* 动态声明。



图 13-1

使用关键字 *new* 时，内存是在堆（heap）中分配的。下面这段代码在栈中创建了一个变量 *ptr*，然后在堆中分配 *ptr* 所指向的内存。

```
int* ptr;  
ptr = new int;
```

图 13-2 显示了执行这两行代码之后内存的状态。注意，即使变量 *ptr* 指向堆中的内存，但是它仍然存在于栈中。指针只是一个变量，它可以存在于栈中，也可以存在于堆中，不过这一点很容易被忘记。然而，动态内存总是在堆中分配的。



图 13-2

下面这个例子说明，指针既可以存在于栈中，也可以存在于堆中。

```
int** handle;  
  
handle = new int*;  
*handle = new int;
```

上面这段代码首先声明了一个指针作为变量 *handle*，这个指针指向一个整数指针。然后动态分配了足够的内存空间来保存整数指针，并把该整数指针存储在 *handle* 的新内存中。接着，向 *\*handle* 的内存空间赋值一个指针，这个指针指向另一段动态内存，这段内存有足够大的空间来保存整数。图 13-3 显示了这两个层次的指针，一个指针存在于栈中（*handle*），另一个存在于堆中（*\*handle*）。

“句柄”（handle）一词有时候用于描述这样一个指针，它指向的也是一个指针，后者指向某个内存单元。在一些应用中，之所以使用句柄，是因为句柄允许底层软件在必要时移动内存。句柄在这里的用法比在第 5 章中更具体，不过所遵循的原则还是一样的，同样是通过一个间接层来进行访问。

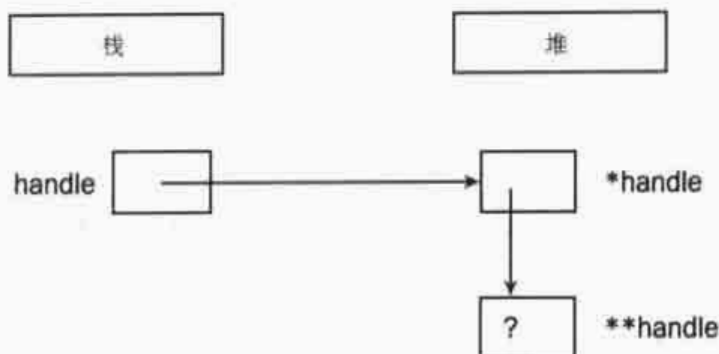


图 13-3

### 13.1.2 内存的分配与撤销

从本书前面的一些章节中，你应该已经熟悉了动态内存的基本知识。为变量创建空间要使用关键字 `new`。要释放该内存空间，以便由程序的其他部分使用，则使用关键字 `delete`。当然，如果 `new` 和 `delete` 这样的简单概念没有什么复杂的变化，那就不是 C++ 了。

#### 使用 `new` 和 `delete`

在本书前面的章节以及这一章前面的介绍中，你已经看到了使用 `new` 的一些最常见的方法。需要分配内存块时，可以基于需要内存空间的变量的类型来调用 `new`。`new` 会返回一个指针，指向分配好的内存，但是要把这个指针存储在一个变量中，这个工作则要你自己来做。如果忽略了 `new` 的返回值，或者指针变量超出了作用域，那么内存就会变成孤立内存（orphaned），因为你再无法访问这块内存空间了。

比如，下面这段代码就使得保存一个 `int` 变量的内存变成孤立内存。图 13-4 显示了执行这段代码之后内存的状态。堆中的数据块无法由栈进行访问（不管是直接访问还是间接访问），所以这块内存就是孤立内存了。

```
void leaky()
{
    new int; // BUG! Orphans memory!
    cout << "I just leaked an int!" << endl;
}
```



图 13-4

除非有办法让计算机提供无限的快速内存，否则就要告诉编译器，什么时候可以释放与对象关联的内存空间，来用于其他的用途。要释放堆中的内存，只要基于指向内存的指针使用关键字 `delete` 就行了，如下所示。

```
int* ptr;
ptr = new int;
```

```
delete ptr;
```

作为一条经验，使用 new 分配内存的每一行代码都应该对应于使用 delete 释放相同内存的另一行代码。

### 使用熟悉的 malloc 会怎么样？

如果你是一名 C 程序员，可能就会产生疑问，为什么不用函数 malloc()，它有什么不好吗？在 C 中，malloc() 用于分配给定字节数的内存空间。对于大多数代码，使用 malloc() 简单而直观。C++ 中也有函数 malloc()，但是建议尽量不要使用。相对于 malloc()，new 的主要优点是，new 不仅仅会分配内存，它还会构造对象。

例如，请考虑下面这两行代码，它使用了一个假想的类 Foo：

```
Foo* myFoo = (Foo*)malloc(sizeof(Foo));

Foo* myOtherFoo = new Foo();
```

在执行完这两行代码之后，myFoo 与 myOtherFoo 都会指向堆中的内存区，这两个内存区足以容纳 Foo 对象。Foo 的数据成员和方法可以使用这两个指针来访问。区别就是，myFoo 指向的 Foo 对象并不是正确的对象，因为从来没有构造过 myFoo。malloc() 函数仅仅是预留一块固定大小的内存。它并不知道也不关心对象是什么。相比而言，new 调用会分配合适大小的内存空间，而且还会正确地构造对象。第 16 章会更详细地解释 new 的这两项任务。

free() 函数和 delete 函数之间也存在类似的差别。使用 free() 时，不会调用对象的析构函数。而使用 delete 时，会调用对象的析构函数，并且会正确清除该对象。

不要把 malloc() 和 free() 与 new 和 delete 混淆。我们建议只使用 new 和 delete。

### 内存分配失败时

许多（甚至大多数）程序员在写程序时都会认为 new 总是成功的。对此有一个原则：如果 new 失败了，就意味着内存不够，这是非常糟糕的。这常常会导致不可知的状态，因为并不清楚程序在这种情况下会怎么办。

默认的情况是，如果 new 操作失败，程序就会终止。在许多程序中，这种行为是可接受的。new 失败时，程序会退出，因为如果没有足够的内存空间来处理请求的话，new 会抛出一个异常。第 15 章会介绍内存不足的情况下妥善恢复的几种方法。

还有另外一个版本的 new，它不会抛出异常，而只是返回一个 NULL，这类似于 C 中的函数 malloc() 的行为。使用这个 new 的语法如下所示：

```
int* ptr = new (nothrow) int;
```

当然，与抛出异常的 new 一样，使用这个版本的 new 时（即只返回 NULL），仍然有同样的问题——如果结果是 NULL 该怎么办？编译器不要求检查结果，所以返回 NULL 的 new（即 nothrow 版本的 new）比抛出异常的 new 更可能导致另外一些 bug。由于这个原因，我们建议还是使用标准的 new。在程序中，

如果从内存不足的情况中恢复很重要，那么可以参考第 15 章，其中会介绍你需要的各种工具。

### 13.1.3 数组

数组把多个相同类型的变量包装在一个带索引的变量中。对于新手程序员，很快就能使用数组，这是很自然的，因为可以很容易地把数组看作是将值置于带有编号的槽中。数组在内存中的表示和这个思维模型也相距不远。

#### 基本类型数组

程序给数组分配内存时，它会分配连续（contiguous）的内存段，其中的每一段都足以保存数组的单个元素。比如，包含 5 个 int 元素的数组可以如下在栈中声明：

```
int myArray [5];
```

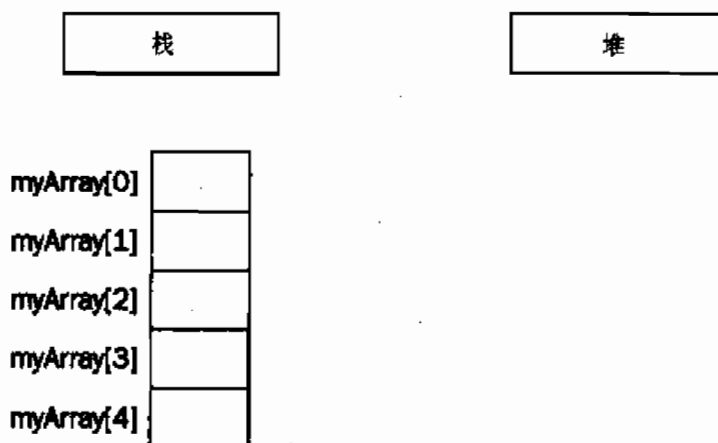


图 13-5

图 13-5 显示了声明数组之后内存的状态。在堆上声明数组没有什么区别，只是要使用指针来指示数组的位置。下面这段代码为包含 5 个 int 元素的数组分配了内存，并用变量 myArrayPtr 存储指向该段内存的指针。

```
int* myArrayPtr = new int [5];
```

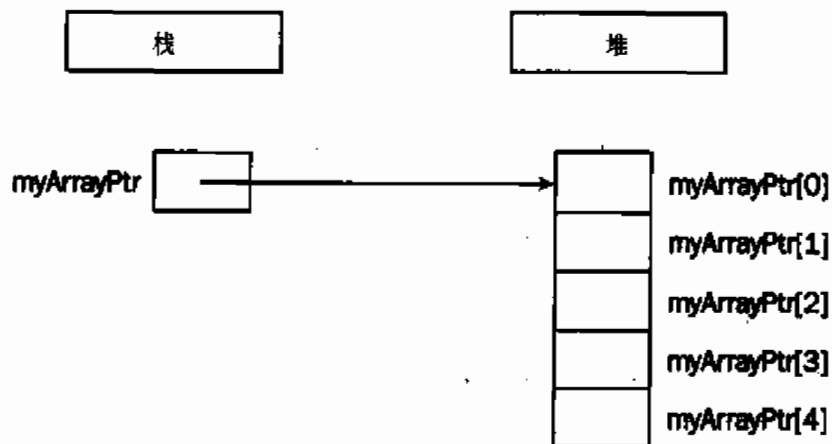


图 13-6

如图 13-6 所示，基于堆的数组类似于基于栈的数组，但是它们在内存中的位置不同。变量 `myArrayPtr` 指向数组的第 0 个元素。把数组放在堆中的优点是，可以使用动态内存存在运行时定义数组的长度。比如，下面这个函数从一个假想的函数 `askUserForNumberOfDocuments()` 接收指定数量的文档，并使用该结果创建一个 `Document` 对象数组。

```
Document* createDocArray()
{
    int numDocs = askUserForNumberOfDocuments();

    Document* docArray = new Document[numDocs];

    return docArray;
}
```

一些编译器通过神秘的方法，允许在栈上建立长度可变的数组。这不是 C++ 的标准特性，遇到这种情况时，建议还是要慎重，应当避免使用。

在前面这个函数中，`docArray` 是动态分配的数组。不要把它和动态数组（dynamic array）混淆。这个数组本身不是动态的，因为一旦分配，数据的长度就不会改变。动态内存允许你在运行时指定分配块的大小，但不是自动地调整其大小来适应数据。确实有一些能动态调整大小来适应数据的数据结构，比如 STL 内置类 `vector`。

C++ 中有一个函数 `realloc()`，它是从 C 语言中保留下来的。不要使用这个函数！在 C 中，函数 `realloc()` 会按新容量分配一个内存块，把所有旧的数据移到新的位置，从而有效地改变数组的长度。这个方法在 C++ 中很危险，因为用户定义的对象不能很好地进行按位复制。

不要在 C++ 中使用函数 `realloc()`，它可不是你的朋友。

### 对象数组

对象数组与基本类型的数组并没有什么区别。使用 `new` 分配 `N` 个对象的数组时，会分配足够的内存空间，其中包括 `N` 个连续的块，每个块都足够大，足以存放单个对象。使用 `new` 会对各个对象自动调用无参数的构造函数。这样一来，使用 `new` 分配对象数组就会返回一个指针，它指向一个充分构建并已初始化的对象数组。

例如，请考虑下面这个类：

```
class Simple
{
public:
    Simple() { cout << "Simple constructor called!" << endl; }
};
```

如果要分配一个包含 4 个 `Simple` 对象的数组，会调用 4 次 `Simple` 构造函数：

```
int main(int argc, char** argv)
{
    Simple* mySimpleArray = new Simple[4];
}
```

这段代码的输出结果如下：

```
Simple constructor called!  
Simple constructor called!  
Simple constructor called!  
Simple constructor called!
```

这个数组的内存图如图 13-7 所示。可以看到，它与基本类型的数组并没有什么区别。

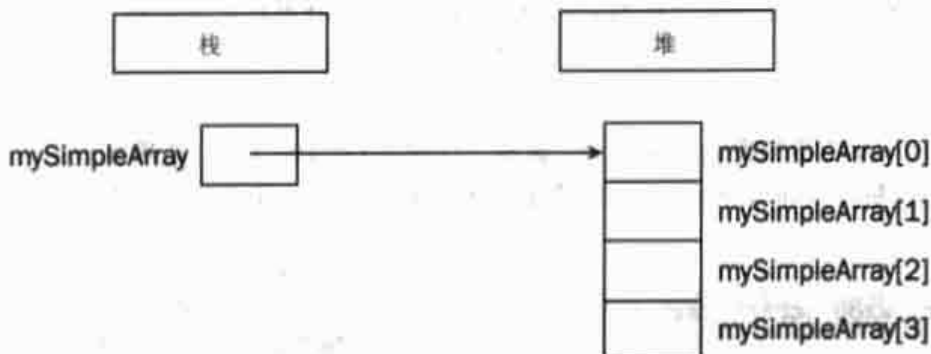


图 13-7

### 删除数组

使用用于数组的 `new` (`new[]`) 分配内存时，就必须用用于数组的 `delete` (`delete[]`) 来释放内存。这个 `delete` 除了释放数组相关的内存外，还会自动地撤销数组中的对象。如果没有使用用于数组的 `delete`，程序可能会有奇怪的表现。在一些编译器中，只会调用数组中第 0 个元素的析构函数，因为编译器只知道你在删除指向一个对象的指针。在另外一些编译器中，可能会发生内存破坏，因为 `new` 和 `new[]` 可能使用完全不同的内存分配机制。

```
int main(int argc, char** argv)
{
    Simple* mySimpleArray = new Simple[4];

    // Use mySimpleArray.

    delete[] mySimpleArray;
}
```

当然，只有当数组中的元素是纯对象时才会调用析构函数。如果有一个指针数组，则仍然需要单独地删除各个元素，就像单独分配每个元素一样，如下面这段代码所示：

```
int main(int argc, char** argv)
{
    Simple** mySimplePtrArray = new Simple*[4];

    // Allocate an object for each pointer.
    for (int i = 0; i < 4; i++) {
        mySimplePtrArray[i] = new Simple();
    }

    // Use mySimplePtrArray.

    // Delete each allocated object.
    for (int i = 0; i < 4; i++) {
        delete mySimplePtrArray[i];
    }
}
```



```
    }  
    // Delete the array itself.  
    delete[] mySimplePtrArray;  
}
```

不要把 new 和 delete 与 new[] 和 delete[] 相混淆，它们是分别配对使用的。

### 多维数组

多维数组扩展了索引值的概念来使用多个编号。比如，Tic-Tac-Toe（连珠游戏）可能会使用二维数组来表示  $3 \times 3$  的网格。下面这个例子显示了如何在栈上声明这样一个数组，并通过一些测试代码来访问该数组。

```
int main(int argc, char** argv)  
{  
    char board[3][3];  
  
    // Test code  
    board[0][0] = 'X'; // X puts marker in position (0,0).  
    board[2][1] = 'O'; // O puts marker in position (2,1).  
}
```

你可能想知道，二维数组中的第一个下标到底是 x 坐标还是 y 坐标。事实上这没有什么关系，只要保持一致即可。 $4 \times 7$  的网格可以声明为 `char board[4][7]` 或者 `char board[7][4]`。对于大部分应用来说，更容易把第一个下标看作 x 轴而把第二个下标看作 y 轴。

### 多维栈数组

在内存中，基于栈的二维数组看起来就如图 13-8 所示。由于内存不能有两个坐标轴（地址只是在一维上连续），所以计算机只能像表示一维数组那样表示二维数组。其区别在于数组的大小和访问数组所用的方法有所不同。

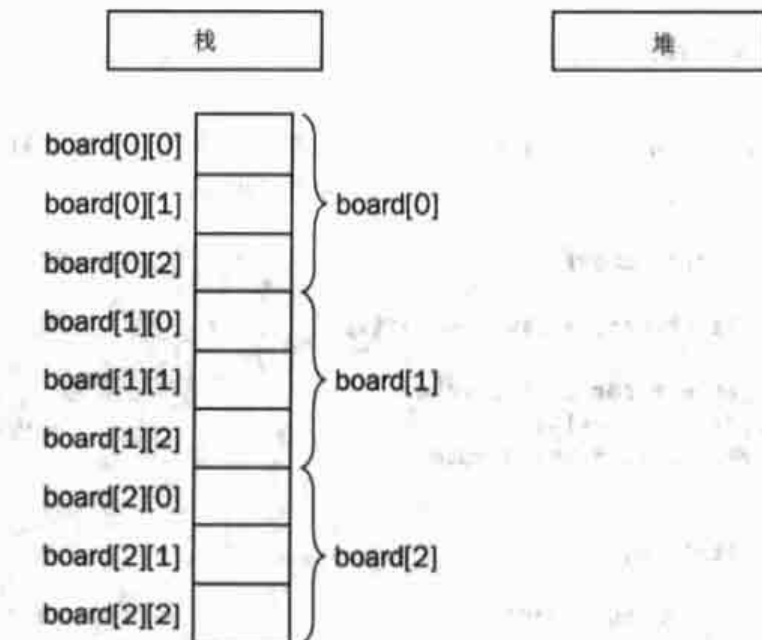


图 13-8

多维数组的大小是各维大小的乘积,然后再乘以数组中单个元素的大小。在图 13-8 中,假设每个字符的大小为 1 个字节,那么  $3 \times 3$  的棋盘的大小就是  $3 \times 3 \times 1 = 9$  字节。对于  $4 \times 7$  的字符棋盘,数组大小则为  $4 \times 7 \times 1 = 28$  字节。

要访问多维数组中的值,计算机把每个下标作为访问多维数组中另一个子数组来处理。比如,在  $3 \times 3$  的网格中,表达式 `board[0]` 实际上是指图 13-9 中突出显示的子数组。增加第二个下标时,比如 `board[0][2]`,计算机就可以通过查找子数组中的第二个下标来访问相应的元素,如图 13-10 所示。

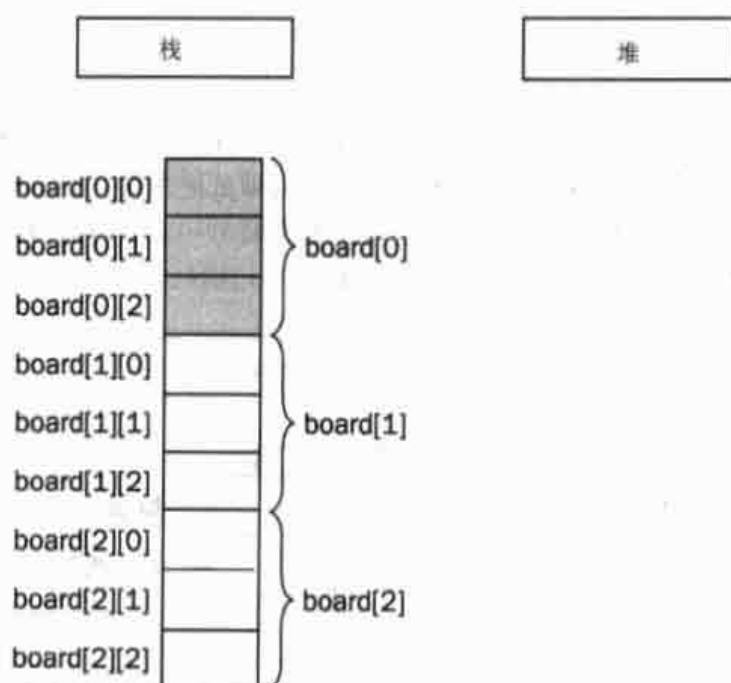


图 13-9

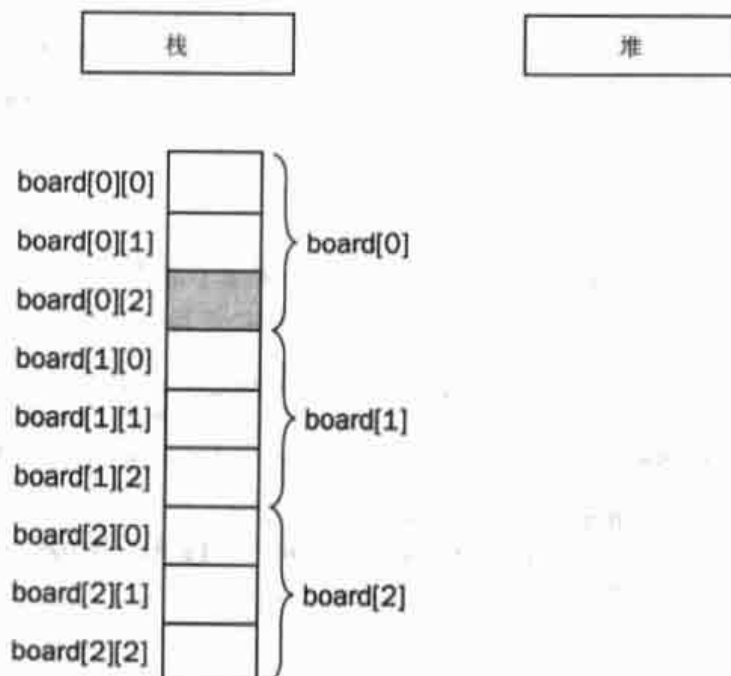


图 13-10

上面介绍的这些技术还可以扩展到 N 维数组，不过高于三维的数组很难概念化，而且在日常的应用中也很少使用。

### 多维堆数组

如果需要在运行时确定多维数组的维数，可以使用基于堆的数组。就像通过指针访问动态分配的一维数组一样，动态分配的多维数组也可以通过指针访问。惟一的区别是，在二维数组中，需要利用指向指针的指针，而在多维数组中，则需要 N 层指针。乍一看，下面这样声明和分配一个动态分配的多维数组似乎是正确的。

```
char** board = new char[i][j]; // BUG! Doesn't compile
```

这段代码不能编译通过，因为基于堆的数组不像基于栈的数组那样工作。为其分配的内存不是连续的，所以给基于堆的多维数组分配足够的内存是不对的。正确的做法应当是，必须先为基于堆的数组的第一维下标分配一个连续的数组。该数组的每个元素实际上是指向另一个数组的指针，这个数组存储了对应第二维下标的元素。这种  $2 \times 2$  的动态分配棋盘布局如图 13-11 所示。

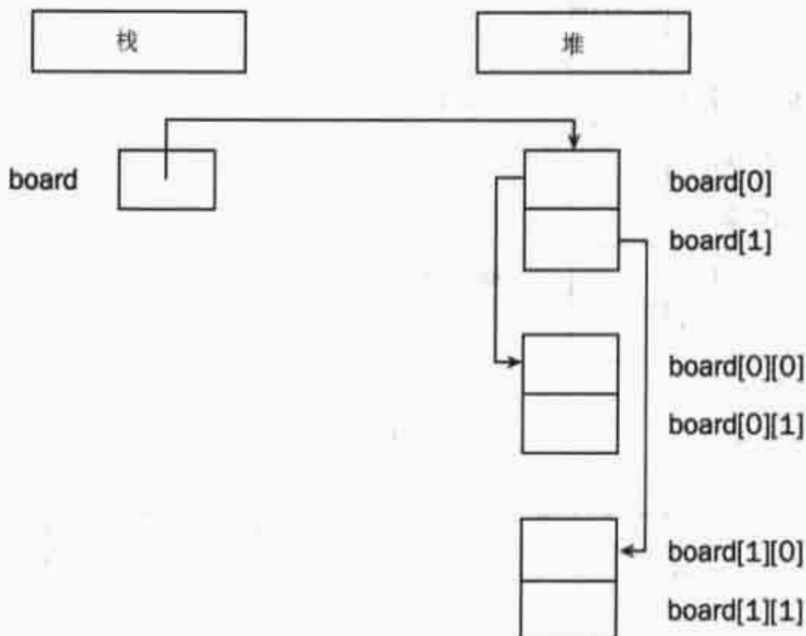


图 13-11

遗憾的是，编译器不会给子数组分配内存。你可以向分配基于堆的一维数组那样先分配第一维的数组，但是各个子数组必须明确分配。下面这个函数会给二维数组适当地分配内存空间。

```
char** allocateCharacterBoard(int xDimension, int yDimension)
{
    char** myArray = new char*[xDimension]; // Allocate first dimension

    for (int i = 0; i < xDimension; i++) {
        myArray[i] = new char[yDimension]; // Allocate ith subarray
    }

    return myArray;
}
```

需要释放与基于堆的多维数组相关联的内存时，用于数组的 `delete[]` 语法并不会为你删除子数组。释放数组的代码应该与分配该数组的代码相对应，如下面的函数所示。

```
void releaseCharacterBoard(char** myArray, int xDimension)
{
    for (int i = 0; i < xDimension; i++) {
        delete[] myArray[i];    // Delete ith subarray
    }

    delete[] myArray;          // Delete first dimension
}
```

### 13.1.4 使用指针

由于指针使用相对容易，所以很可能会遭到滥用，这就导致指针有一个不好的名声。因为指针仅仅是一个内存地址，在理论上可以手工改变该地址，甚至会像下面这行代码一样做一些不该做的事情：

```
char* scaryPointer = 7;
```

上面这行代码建立了一个指向内存地址 7 的指针，它可能是一个随机垃圾，也可能是应用中其他地方正在使用的一个内存。如果使用了不是用 `new` 分配的内存区域，最终会破坏与对象关联的内存，程序也会崩溃。

#### 指针的思维模型

在第 1 章已经读到，可以从两个角度理解指针。在数学思维方面较强的读者可能会把指针简单地看作一个地址。基于这个观点，可以更容易地理解指针运算，这会在本章后面介绍。指针并不是通向内存的神秘通道，它只是一些数字，碰巧与内存中的某个位置对应而已。图 13-12 展示了基于地址观点的  $2 \times 2$  的网格。

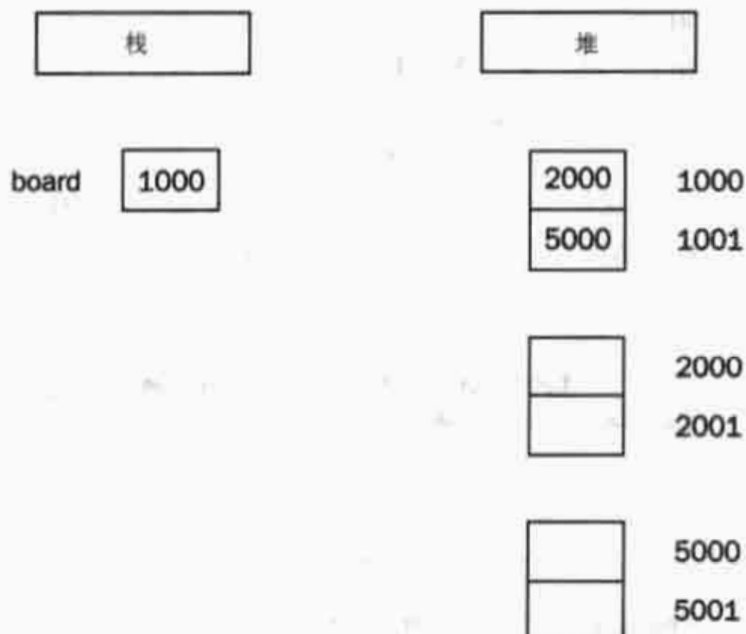


图 13-12

熟悉空间表示方法的读者可能更容易理解指针的“箭头”观点。指针只是一个间接层，它告诉程序“嘿！看这里”。从这个观点来看，多层指针实际上只是访问数据过程中的各个步骤。图 13-11 给出了指针在内存中的图示视图。

使用 \* 操作符对指针解除引用 (dereference) 时，是告诉程序要在内存中再深入一层。采用基于地址的观点，可以把解除指针引用看作是在内存中跳到指针指示的地址。从图形的观点来看，每次解除指针引用都对应于沿着指针箭头从基址转到其指向的地址。

取变量地址要用 & 操作符，这是在内存中加了一个间接层。按照基于地址的观点，程序只是指出了变量的地址，它可以存储在指针变量中。在图形观点中，操作符 & 创建了一个新的箭头，它的终点就是该变量（即指向该变量）。箭头的基址可以附加到一个指针变量上。

### 指针类型强制转换

既然指针仅仅是内存地址（或者指向某个位置的箭头），所以指针是弱类型的。指向 XML 文档的指针与指向整数的指针大小也是一样的。通过使用 C 风格的类型强制转换，编译器可以很容易地把任何指针类型强制转换为另一种指针类型。

```
Document* documentPtr = getDocument();  
char* myCharPtr = (char*)documentPtr;
```

static 类型强制转换的安全性更高些。编译器会拒绝对指向不同数据类型的指针完成 static 类型强制转换。

```
Document* documentPtr = getDocument();  
static_cast<char*>(documentPtr); // BUG! Won't compile
```

如果要强制转换类型的两个指针实际上是指向通过继承相关联的对象，编译器会允许完成 static 类型强制转换。但是，就像在第 10 章介绍的那样，在继承层次结构中，动态的类型强制转换更为安全。

### const 指针

关键字 const 和指针之间的相互作用有点让人糊涂，因为你不清楚在给谁应用 const。假设动态分配了一个整数数组，并为之应用了 const，那么是数组的地址受 const 保护，还是数组中的各个值受 const 保护？这个问题的答案要依赖于使用了怎样的语法。

如果 const 位于类型之前，它表示指针指向的值是受保护的。在数组中，就表明数组的各个元素是 const 的。下面这个函数接收一个指向 const 整数的指针。第一行不能编译，因为实际的值是受 const 保护的。第二行代码能够编译，因为数组本身并没有受到 const 的保护。

```
void test(const int* inProtectedInt, int* anotherPtr)  
{  
    *inProtectedInt = 7; // BUG! Attempts to write to read-only value  
    inProtectedInt = anotherPtr; // Works fine  
}
```

要保护指针本身，关键字 const 要直接位于变量名的前面，如下面这段代码所示。这时，指针和指针指向的值就都是受保护的，所以这两行代码都无法编译通过。

```
void test(const int* const inProtectedInt, int* anotherPtr)  
{  
    *inProtectedInt = 7; // BUG! Attempts to write to read-only value  
    inProtectedInt = anotherPtr; // BUG! Attempts to write to read-only value  
}
```

在实际中，很少需要保护指针。如果函数能够改变所传递的指针值，这也无关大碍。其作用仅限于这个函数内部，对于调用者而言，指针仍然指向它原来的地址。把指针设置成 `const`，这对于说明指针的用途更有意义，其实提供不了多少真正的保护。但是，保护指针指向的值则很常见，从而防止重写共享数据。

## 13.2 数组与指针的对应

你已经看到了，指针和数组之间存在某种重叠。在堆上分配的数组由指向第一个元素的指针来引用。基于栈的数组使用数组语法 (`[]`) 来引用，这种语法采用一种常规的变量声明。然而，就像下面将要介绍的，这种重叠并非到此为止。指针和数组之间还有一种更复杂的关系。

### 13.2.1 数组即指针

基于堆的数组不是惟一用指针引用数组的地方。也可以使用指针的语法来访问基于栈的数组中的元素。数组的地址其实是第 0 个元素的地址。编译器知道，当通过变量名来整个地引用数组时，实际上是引用第 0 个元素的地址。采用这种方式，指针就像是一个基于堆的数组一样进行工作。下面这段代码在栈上创建了一个数组，但是使用了指针来访问该数组。

```
int main(int argc, char** argv)
{
    int myIntArray[10];

    int* myIntPtr = myIntArray;

    // Access the array through the pointer.
    myIntPtr[4] = 5;
}
```

向函数传递数组时，能够通过指针来引用基于栈的数组，这是很有用的。下面这个函数接受整数数组作为指针参数。需要注意，调用者需要明确地传入数组的大小，因为从指针无法了解数组大小。实际上，任何形式的 C++ 数组（无论是否是指针），都没有内置数组大小的概念。

```
void doubleInts(int* theArray, int inSize)
{
    for (int i = 0; i < inSize; i++) {
        theArray[i] *= 2;
    }
}
```

这个函数的调用者可以传递一个基于栈的数组或者基于堆的数组。在传递基于堆的数组时，指针已经存在，只是简单地按值传递给函数即可。在传递基于栈的数组时，调用者可以传递数组变量，编译器会自动地把数组变量当作是指向数组的指针来处理。这两种用法如下所示：

```
int main(int argc, char** argv)
{
    int* heapArray = new int[4];
    heapArray[0] = 1;
    heapArray[1] = 5;
    heapArray[2] = 3;
    heapArray[3] = 4;
```



```
doubleInts(heapArray, 4);

int stackArray[4] = {5, 7, 9, 11};

doubleInts(stackArray, 4);
}
```

即使函数没有显式的指针参数，但数组作为参数传递的语义与指针有着惊人的相似性，因为把数组传递给函数时，编译器会把数组作为指针处理。如果一个函数取数组作为实参，并改变了数组中的值，这个函数实际上修改的是原始数组，而不是数组的副本。就像指针一样，传入数组可以有效地模拟传引用功能，因为实际传递给函数的是原始数组的地址，而不是数组的副本。函数 `doubleInts()` 的以下实现中，即使参数是数组，不是指针，它也会改变原始数组。

```
void doubleInts(int theArray[], int inSize)
{
    for (int i = 0; i < inSize; i++) {
        theArray[i] *= 2;
    }
}
```

你可能会疑问，为什么会这样呢？函数定义中明明使用了数组语法，为什么编译器不只是复制数组呢？一个可能的解释就是效率——复制数组的元素需要花费时间，还很有可能消耗大量内存空间。通过传递指针，编译器就不需要包含复制数组的代码了。

下面做一个总结。使用数组语法声明的数组可以通过指针来访问。把数组传递给函数时，总是作为指针传递的。

### 13.2.2 指针并非都是数组

由于编译器允许在希望传入指针的地方传入数组，如前面的函数 `doubleInts()` 所示，这可能会让你认为，指针和数组是一样的。实际上，这二者之间存在着微妙但很重要的区别。指针和数组有许多共同的性质，有时候可以交替使用（如前面的例子所示），但是它们并不完全相同。

指针自身并没有什么意义。它可以指向随机的内存、单个对象或者数组。可以一直对指针使用数组语法，但是这样做并不总是正确的，因为指针并非都是数组。比如，看一下下面这行代码：

```
int* ptr = new int;
```

指针 `ptr` 是一个合法的指针，但不是数组。可以使用用于数组的语法（`ptr[0]`）来访问指针指向的值，但是这样做在文法上是有问题的，并没有什么真正的好处。实际上，对非数组指针使用数组语法往往会带来 bug。`ptr[1]` 处的内存可以是任何值！

数组会自动引用为指针，但是并非所有的指针都是数组。

## 13.3 动态字符串

字符串给程序设计语言的设计者带来了一个难题，字符串看起来像是一个标准数据类型，但又不能表述为固定大小。不过，字符串用的如此之多，以至于大多数程序设计语言都需要有一个内置的字符串模型。在 C 语言中，字符串有点像外来户，从来没有得到本应作为首类语言特性而受到的关注。C++ 则

提供了更灵活、更有用的字符串表示。

### 13.3.1 C 风格的字符串

在 C 语言中，字符串表示为字符数组。字符串的最后一个字符是 null 字符（‘\0’），这样，对字符串进行操作的代码可以确定字符串在什么地方结束。尽管 C++ 提供了一个更好的字符串抽象，但理解 C 语言的字符串技术也是很重要的，因为它还会在 C++ 程序设计中出现。

迄今为止，程序员使用 C 字符串时最常见的错误是，他们常常忘记给字符 ‘\0’ 分配内存空间。例如，字符串 “hello” 看起来长度为 5 个字符，但是在内存中需要 6 个字符串长度的内存空间来存储该值，如图 13-13 所示。

C++ 包含了来自 C 语言的几个对字符串进行操作的函数。作为一条一般经验，这些函数不处理内存分配。例如，函数 strcpy() 取两个字符串作为参数。它把第二个字符串复制到第一个字符串，不管前者是否能容纳得下后者。下面这段代码试图把函数 strcpy() 包装起来，它要分配大小合适的内存并返回结果，而不是把结果放在一个已分配的字符串中。这个“包装器”函数使用了函数 strlen() 来获取字符串的长度。

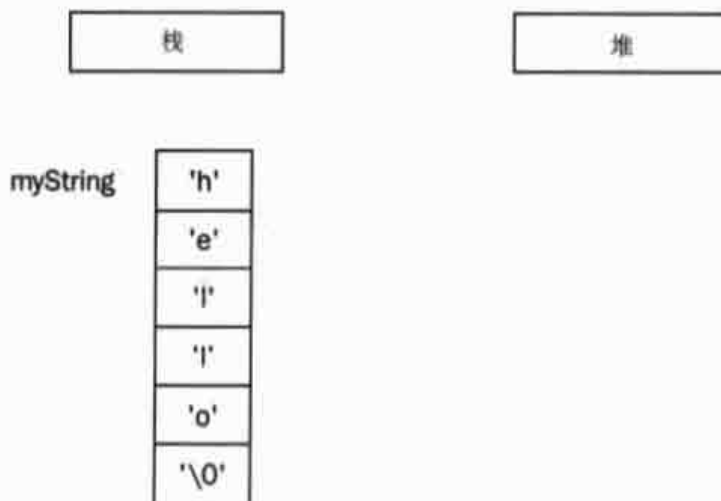


图 13-13

```
char* copyString(const char* inString)
{
    char* result = new char[strlen(inString)]; // BUG! Off by one!

    strcpy(result, inString);

    return result;
}
```

上面编写的函数 copyString() 是不正确的。函数 strlen() 返回的是字符串的长度，而不是保存字符串所需的内存大小。对于字符串 “hello”，函数 strlen() 返回 5，而不是 6。给字符串分配内存时，正确的方法是在实际字符所需的内存空间之外再加 1。乍一看加 1 有点古怪，但是很快你就会发现这是很自然的，如果没有加 1，（但愿）你倒会觉得有问题。

```
char* copyString(const char* inString)
{
    char* result = new char[strlen(inString) + 1];

    strcpy(result, inString);

    return result;
}
```

函数 `strlen()` 只是返回字符串中实际的字符个数，要记住这一点，一个方法就是考虑一下，如果正在给一个字符串分配空间，而这个字符串由其他几个字符串组成，此时会是什么情况。例如，如果函数有三个字符串参数，并会返回一个字符串，作为返回值的字符串是这三个字符串参数连接的结果，那么它会有多大？要保留刚刚好的内存空间，那就是三个字符串的长度加在一起，然后再加 1（这对应为在结尾再加上字符 `'\0'`）。如果函数 `strlen()` 返回的字符串长度中包含了 `'\0'` 对应的额外的 1，分配的内存空间就会过大。下面这段代码使用函数 `strcpy()` 和 `strcat()` 来完成这个操作。

```
char* appendString(const char* inStr1, const char* inStr2, const char* inStr3)
{
    char* result = new char[strlen(inStr1) + strlen(inStr2) + strlen(inStr3) + 1];

    strcpy(result, inStr1);
    strcat(result, inStr2);
    strcat(result, inStr3);

    return result;
}
```

对字符串进行操作的 C 函数的完整列表请见 `<cstring>` 头文件。

### 13.3.2 字符串直接量

你可能已经见过在 C++ 程序中编写的带引号的字符串。比如，下面这行代码通过包含 `hello` 字符串自身来将其输出，而没有使用包含这个字符串的变量。

```
cout << "hello" << endl;
```

在上面这行代码中，“hello”是一个字符串直接量（string literal，也称字符串字面量），因为它作为一个值而不是一个变量。即使字符串直接量没有相关联的变量，仍是将其处理为 `const char*`（常量字符数组）。

字符串直接量可以赋值给变量，但是这样做会有风险。与字符串直接量相关联的具体内存空间位于内存的只读部分，这就是为什么它是常量字符数组的原因。这就允许编译器通过重用指向等价字符串直接量的引用来优化内存的使用（也就是说，即使程序使用了字符串直接量“hello”500 次，编译器也只是在内存中创建 `hello` 的一个实例）。但是编译器并不强制程序只是把字符串直接量赋值给 `const char*` 类型或者 `const char[]` 类型的变量。也可以把字符串赋值给没有加 `const` 的 `char*` 类型变量，除非你试图改变这个字符串，否则程序都能很好地工作。通常，如果试图改变字符串，这会立即使程序崩溃，就像下面这段代码所示：

```
char* ptr = "hello";           // Assign the string literal to a variable.

ptr[1] = 'a';                  // CRASH! Attempts to write to read-only memory
```

更安全的编码方法是在引用字符串直接量时使用指向 const 字符的指针。下面这段代码也有一个同样的 bug，但是因为这里是把字符串直接量赋值给 const 字符数组，所以编译器会捕获到这种只读内存单元写数据的企图。

```
const char* ptr = "hello"; // Assign the string literal to a variable.  
ptr[1] = 'a';              // BUG! Attempts to write to read-only memory
```

也可以使用字符串直接量作为基于栈的字符数组的初始值。因为基于栈的变量在任何情况下都不可能引用其他地方的内存，所以编译器会负责把字符串直接量复制到基于栈的数组内存中。

```
char stackArray[] = "hello"; // Compiler takes care of copying the array and  
                             // creating appropriate size for stack array  
stackArray[1] = 'a';         // The copy can be modified.
```

### 13.3.3 C++ 的字符串类

前面已经说过，C++ 提供了一个有很大改善的字符串实现，并作为标准库的一部分。在 C++ 中，string 是一个类（实际上是 basic\_string 模板类的一个实例化），它支持 <cstring> 中许多函数所完成的同样的操作，不过，最好的一点是，如果使用得当，string 类会负责分配内存。

#### C 风格的字符串有什么问题

在涉及 C++ 的 string 类之前，先看一下 C 风格的字符串有哪些优缺点。

优点：

- C 风格的字符串比较简单，利用了底层的基本字符类型和数组结构。
- C 风格的字符串占用空间更小，如果正确使用的话，它们只占用真正需要的内存空间。
- C 风格的字符串更底层，所以可以作为原始内存很容易地进行处理和复制。
- C 程序员能够很好地理解——那么为什么还要学习新内容呢？

缺点：

- C 风格字符串不能忍受内存 bug 的存在，而且很受其影响。
- C 风格字符串没有充分利用 C++ 的面向对象特性。
- C 风格字符串提供的辅助函数命名很糟糕，有时还会把人搞糊涂。
- C 风格字符串要求程序员了解字符串的底层表示。

我们精心打造了上面这个列表，目的是使你认为可能还会有更好的做法。下面将学到，C++ 字符串可以解决 C 字符串的所有上述缺点，而且还能使其优点更突出。

#### 使用字符串类

虽然 string 是一个类，但是一般都可以把它看作是一个内置的类型来加以处理，就像 int 一样。实际上，越把它看作是简单类型，就会越轻松。如果能忘记 string 是对象，程序员遇到的与 string 相关的问题就会越少。

基于操作符重载的魔力，C++ 字符串支持使用 + 操作符进行字符串连接，使用 = 操作符进行赋值，用 == 操作符进行比较，用 [] 操作符访问单个的字符。正是因为有这些操作符，才使得程序员可以把 string 看作是一个基本类型。如下面这段代码所示，可以在 string 上完成这些操作，而无需担心内存分配问题。

```
int main(int argc, char** argv)
{
    string myString = "hello";

    myString += ", there";

    string myOtherString = myString;

    if (myString == myOtherString) {
        myOtherString[0] = 'H';
    }

    cout << myString << endl;
    cout << myOtherString << endl;
}
```

这段代码的输出结果如下：

```
hello, there
Hello, there
```

在这个例子中需要注意几点。首先，即使字符串来回分配并调整大小，但是这里不存在内存泄漏。所有这些 string 对象都是作为栈变量创建的。string 类肯定有大量分配和调整大小的操作，不过，如果对象超出作用域，它们会自己清空这些内存。

另外需要注意的一点是，操作符会按照你希望的去工作。你可能会担心，使用操作符=有可能导致两个变量指向同一个内存，但事实并非如此。操作符=会复制字符串，这往往正是你想要的。类似地，操作符==会比较两个字符串的实际内容，而不是比较字符串在内存中的位置。如果习惯于使用基于数组的字符串，这就能让你彻底得到解放，也可能会带来一些困惑。不要担心——一旦了解到可以相信 string 类确实能有得当的表现，就会轻松许多。

由于兼容性，可以使用方法 `c_str()` 把 C++ string 转化为 C 风格的字符串。应该在使用结果之前调用该方法，这样才能准确地反映 string 的当前内容。

在线参考列出了可以对 string 对象完成的所有操作。

## 13.4 低级的内存操作

比起 C 语言来说，C++ 的主要优点之一是不需要特别担心关于内存的问题。如果代码用到了对象，只需要确保各个类能适当地管理它自己的内存即可。通过构造和撤销对象，编译器会告诉你什么时候做什么，从而帮助你管理内存。正如 string 类中所见，把内存管理隐藏在类中，这样能大大提高可用性。

但是对于某些应用，可能会遇到这种情况，即需要在低层次上使用内存。不管是否是由于效率、调试或者好奇等原因，了解一些使用原始字节的技术总是有好处的。

### 13.4.1 指针运算

C++ 编译器使用指针的声明类型来支持完成指针运算 (pointer arithmetic)。如果把指针声明为 int 类型，并将该指针加 1，指针会在内存中根据 int 的大小前移一个单位，而不是移动一个字节。这类操作对于数组最有用，因为它们包含了同构的数据，而且在内存中是顺序排列的。比如，假设要在堆上声明一个 int 数组：



```
int* myArray = new int[8];
```

你已经熟悉下面这个语法，它用来设置第 2 个位置上的值（译者注：即第 3 个元素）：

```
myArray[2] = 33;
```

使用指针运算，可以等价地使用下面的语法，得到一个指针，它指向 myArray “向前移两个 int” 处的内存，然后对指针解除引用来设置值。

```
*(myArray + 2) = 33;
```

作为访问单个元素的备用语，指针运算看起来并不那么吸引人。它真正的力量在于，像 myArray + 2 这样的表达式仍然是指向 int 的指针，因此可以表示一个较小的 int 数组。假设现在有一个 C 风格的字符串，如下所示：

```
const char* myString = "Hello, World!";
```

假设还有一个函数，它的参数是一个字符串，返回结果是一个新的字符串，这个新字符串是把输入的字符串改成大写后的结果。

```
char* toCaps(const char* inString);
```

可以把字符串 myString 传递给这个函数来把它改成大写。但是，若只是想把 myString 的一部分改成大写，则可以使用指针运算来引用字符串后面的部分。下面这段代码对字符串的 World 部分调用函数 toCaps()：

```
toCaps(myString + 7);
```

指针运算的另一个有用的应用涉及到减法。将一个指针减去同类型的另一个指针，所得到的是两个指针之间的元素个数（指定类型），而不是两个指针之间的绝对字节数。

### 13.4.2 自定义内存管理

你所遇到的百分之九十九的情况下（有人可能会说是百分之百），内置的内存分配功能就已经足够了。在后台，new 和 delete 会完成所有的这些工作：以适当大小的块来分配内存，维护可用的内存区列表，删除内存时再把内存块释放到可用内存区列表中。

如果资源要求很紧张，就完全可以自己来管理内存。不要担心——它并不像想像中那么可怕。基本说来，自己管理内存一般是指，由类分配一大块内存，并在需要时将此内存按片进行分发。

这个方法会更好一些吗？自己管理内存可能会减少资源开销。使用 new 分配内存时，程序还需要保留一小块空间来记录已经分配了多少内存空间。这样，调用 delete 时，就可以释放适当数量的内存。对于大部分对象，相对于分配的内存来说，这个开销要小得多，所以并没有太大的区别。然而，对于小的对象或者有大量对象的程序，这种开销可能会有较大的影响。

自己管理内存时，你事先已经知道每个对象的大小，所以可以避免每个对象的开销。对于大量的小对象，与使用 new 和 delete 的方法相比，这样就会带来很大的差别。完成定制的内存管理的有关语法将在第 16 章介绍。

### 13.4.3 垃圾回收

内存管理的另一个方面是垃圾回收。在支持垃圾回收的环境中，程序员很少需要（甚至完全不用）



显式地释放与对象关联的内存。取而代之的是，由一个低优先级的后台任务负责监视内存状态，清理它认为不再需要的内存。

不同于 Java 语言，在 C++ 中，没有把垃圾回收作为内置功能。大部分的 C++ 程序都通过 `new` 和 `delete` 在对象层次上管理内存。在 C++ 中实现垃圾回收也不是不可能，但是要想从释放内存的任务中解脱出来，可能又会带来新的问题。

垃圾回收的一种方法称为标记和清扫 (mark and sweep)。使用这种方法，垃圾回收器会周期性地检查程序中的每个指针，并标记所引用的内存仍在使用。在循环结束时，没有标记的内存就认为未在使用，可以释放。

如果你愿意完成以下步骤，就可以在 C++ 中实现标记和清扫算法：

1. 向垃圾回收器注册所有的指针，这样就可以很容易地扫描整个指针列表。
2. 让所有对象派生一个混合类 (如 `GarbageCollectible`)，它允许垃圾回收器把对象标记为正在使用。
3. 确保垃圾回收器运行时不会对指针做修改，以此来保护对对象的并发访问。

可以看到，这个简单的垃圾回收方法要求程序员很仔细才行。与使用 `delete` 相比，这种方法可能更容易带来错误！在 C++ 中已经试图建立一种安全而容易的机制来完成垃圾回收，但是即使在 C++ 确实提供了一个理想的垃圾回收实现，也不一定适用于所有的应用。垃圾回收存在以下缺点：

- 垃圾回收器主动运行时，可能会使程序的运行减慢。
- 如果程序大量地分配内存，那么垃圾回收器可能跟不上这个速度。
- 如果垃圾回收器本身有 bug，或者认为一个已经抛弃的对象仍然在使用，可能会造成不可恢复的内存泄漏。

#### 13.4.4 对象池

如上面所述，定制内存管理就像在大型的仓储超市里购买野餐用品一样。你在自己的 SUV 里塞满了纸盘子，但现在并不需要这么多，这样的话，下次再去野餐就不用再去超市买了，这样就避免了一次开销。垃圾回收就像把用过的盘子丢到院子里，风很容易把它们吹到邻居家的院子里。当然，可能还有更环保的方法来进行内存管理。

对象池模拟了再利用。就是说，你买的盘子数量是正好的，但是在使用之后把它们再挂起来，这样以后就可以把它们清洗一下重新使用了。如果需要过一段时间使用多个相同类型的对象，而每创建一个对象都存在一定开销，对于这种情况，对象池再合适不过了。

第 17 章会进一步介绍使用对象池来提高性能效率的细节问题。

#### 13.4.5 函数指针

正常情况下不会考虑函数在内存中的位置，但是每个函数的确都位于一个特定的地址。在 C++ 中，可以把函数作为数据使用。换句话说，可以把函数的地址作为参数，就像变量一样使用。

函数指针根据参数类型和兼容函数的返回类型来确定类型。使用函数指针最容易的方法是使用 `typedef` 机制为一组有给定特征的函数赋一个类型名。比如，下面这行代码声明了一个类型 `YesNoFcn`，它表示一个指针，该指针指向有两个 `int` 参数且返回 `bool` 类型的任意函数。

```
typedef bool (*YesNoFcn)(int, int);
```

既然有了这个新类型，就可以编写一个取 `YesNoFcn` 作为参数的函数了。比如，下面这个函数接受两个 `int` 数组和数组大小，还接受一个 `YesNoFcn` 参数。它平行地迭代处理这两个数组，在两个数组的对

应元素上调用 YesNoFcn，如果函数 YesNoFcn 返回 true，则打印一条消息。注意，虽然 YesNoFcn 是作为变量传递的，但是可以像常规的函数一样进行调用。

```
void findMatches(int values1[], int values2[], int numValues, YesNoFcn inFunction)
{
    for (int i = 0; i < numValues; i++) {
        if (inFunction(values1[i], values2[i])) {
            cout << "Match found at position " << i <<
                " (* << values1[i] << ", * << values2[i] << ")* << endl;
        }
    }
}
```

要调用函数 findMatches()，所需要的只是一个遵循已定义的 YesNoFcn 类型的函数——也就是说，只要取两个 int 数组并返回 bool 值的函数都可以。例如，请考虑下面这个函数，如果两个参数相等，它会返回 true：

```
bool intEqual(int inItem1, int inItem2)
{
    return (inItem1 == inItem2);
}
```

因为函数 intEqual() 与 YesNoFcn 类型匹配，所以它可以作为最后的实参传递给函数 findMatches()，如下面这段程序所示：

```
int main(int argc, char** argv)
{
    int arr1[7] = {2, 5, 6, 9, 10, 1, 1};
    int arr2[7] = {4, 4, 2, 9, 0, 3, 4};

    cout << "Calling findMatches() using intEqual(): " << endl;
    findMatches(arr1, arr2, 7, &intEqual);

    return 0;
}
```

注意，函数 intEqual() 是通过取其地址传递给函数 findMatches() 的。理论上讲，字符 & 是可选的——如果只是简单地传递函数名，编译器会知道你的意思是取其地址。这个程序的输出为：

```
Calling findMatches() using intEqual():
Match found at position 3 (9, 9)
```

函数指针的魔力在于：函数 findMatches() 是通用函数，它对两个数组中对应的值进行比较。就像上面的用法，它是基于相等性进行比较。不过，由于它取函数指针作为参数，所以可以根据其他的准则进行比较。例如，下面这个函数也满足 YesNoFcn 的定义。

```
bool bothOdd(int inItem1, int inItem2)
{
    return (inItem1 % 2 == 1 && inItem2 % 2 == 1);
}
```

下面这段程序使用这两个 YesNoFcn 类型 (intEqual 和 bothOdd) 来调用 findMatches()：

```
int main(int argc, char** argv)
{
    int arr1[7] = {2, 5, 6, 9, 10, 1, 1};
    int arr2[7] = {4, 4, 2, 9, 0, 3, 4};

    cout << "Calling findMatches() using intEqual():" << endl;
    findMatches(arr1, arr2, 7, &intEqual);

    cout << endl;

    cout << "Calling findMatches() using bothOdd():" << endl;
    findMatches(arr1, arr2, 7, &bothOdd);

    return 0;
}
```

这段程序的输出如下：

```
Calling findMatches() using intEqual():
Match found at position 3 (9, 9)

Calling findMatches() using bothOdd():
Match found at position 3 (9, 9)
Match found at position 5 (1, 3)
```

通过使用函数指针，可以对一个函数 `findMatches()` 进行定制，使之基于参数 `inFunction` 用于不同的用途。

## 13.5 常见的内存陷阱

要指出在哪些情况下会导致与内存相关的 bug，这是很困难的。每个内存泄漏或者不好的指针都有自己的特异性。要解决内存相关的问题，没有什么银弹，不过存在几类常见的问题，可以使用一些工具来检测并加以解决。

### 13.5.1 字符串空间分配不足

上面已经介绍过，C 风格的字符串最普遍的问题是没有分配足够的空间。在大部分情况下，都是因为程序员没有为最后的结束标识字符 `'\0'` 分配另外一个字符空间，所以导致了这个问题。程序员假设字符串有一个固定的最大长度时，也会产生字符串空间分配不够。基本的内置字符串函数不会受固定长度的限制——它们很有可能把字符串的最后部分写到固定大小之外的内存中。

下面这段代码从网络连接中读取数据，并放到一个 C 风格的字符串中。因为网络连接一次只能接收很少数量的数据，所以这里使用循环来完成这个工作。函数 `getMoreData()` 返回 `NULL` 时，就已经接收了所有的数据。

```
char buffer[1024]; // Allocate a whole bunch of memory.
bool done = false;

while (!done) {
    char* nextChunk = getMoreData();
    if (nextChunk == NULL) {
        done = true;
    } else {
```

```
strcat(buffer, nextChunk); // BUG! No guarantees against buffer overrun!  
delete[] nextChunk;
```

解决这个问题有三种途径。按优先顺序，这三种方法分别是：

1. 找一个特定版本的 `getMoreData()` 函数，它取最大大小作为参数。每次调用 `getMoreData()` 时，只给它提供所有剩余的内存空间作为最大大小。
2. 跟踪在缓冲区中还剩余多少内存空间。当不足以容纳当前的块时，分配一块两倍大的新缓冲区，并把原缓冲区复制到这个新缓冲区中。
3. 使用 C++ 风格的字符串，它会为你处理与字符串连接相关联的内存问题。这是个好办法，就用它了！

### 13.5.2 内存泄漏

查找并修复内存泄漏是 C++ 程序设计过程中更令人畏惧的一个部分了。程序可能最后能够运行，并且看起来可以给出正确的结果。接下来你开始注意到，程序运行时会“吞噬”越来越多的内存。那么程序肯定存在内存泄漏。

如果分配了内存，但是忘了释放，此时就会产生内存泄漏问题。首先，这听起来好像只是程序设计时不够细心，似乎很容易避免。毕竟，只要编写的每个类中每个 `new` 都对应于一个 `delete`，就应该没有内存泄漏问题了，是这样吗？实际上，并不总是如此。在下面这段代码中，类 `Simple` 得到了正确的编写，它确实释放了所分配的所有内存。但是，调用 `doSomething()` 函数时，指针会改为另一个 `Simple` 对象，但原来的 `Simple` 对象并没有删除。一旦丢掉了指向对象的指针，就几乎无法再删除这个对象了。

```
#include <iostream>  
  
using namespace std;  
  
class Simple  
{  
public:  
    Simple() { mIntPtr = new int(); }  
    ~Simple() { delete mIntPtr; }  
  
    void setIntPtr(int inInt) { *mIntPtr = inInt; }  
    void go() { cout << "Hello there" << endl; }  
  
protected:  
    int* mIntPtr;  
};  
  
void doSomething(Simple*& outSimplePtr)  
{  
    outSimplePtr = new Simple(); // BUG! Doesn't delete the original.  
}  
  
int main(int argc, char** argv)  
{  
    Simple* simplePtr = new Simple(); // Allocate a Simple object.  
    doSomething(simplePtr);  
}
```



```
delete simplePtr; // Only cleans up the second object
}
```

在上面这个例子的情况下，内存泄漏可能是由于程序员之间交流不畅所导致，或者是由于没有建立很好的代码文档造成的。函数 `doSomething()` 的调用者可能并没有意识到变量是按引用传递的，因此不可能想到会对指针重新赋值。如果确实注意到参数是指针的一个非 `const` 引用，可能就会怀疑发生了什么奇怪的事情，但是函数 `doSomething()` 没有提供注释来解释这个行为。

### 内存泄漏的查找与修复

内存泄漏很难跟踪到，因为要想查看内存，看哪些对象没有使用，以及这些对象原来是在什么地方分配的，做到这一点并不容易。幸运的是，已经有一些程序可以做这些事情。内存泄漏检测工具有很多，从昂贵的专业软件包到免费的可下载工具都有。`valgrind` 就是这样一个免费工具，这是一个面向 Linux 的开源工具，它可以准确找出分配泄漏对象的那一行代码，除此以外还有许多其他功能。

下面这个输出结果就是在前一段程序上运行 `valgrind` 生成的，它准确地指出分配了内存但是从来没有释放的确切位置。在这种情况下，有两处内存泄漏——第一个是 `Simple` 对象没有删除，第二个是它创建的基于堆的整数未删除。

```
==15606== Memcheck, a.k.a. Valgrind, a memory error detector for x86-linux.
==15606== Copyright (C) 2002-2003, and GNU GPL'd, by Julian Seward.
==15606== Using valgrind-2.0.0, a program supervision framework for x86-linux.
==15606== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
==15606== Estimated CPU clock rate is 1136 MHz
==15606== For more details, rerun with: -v
==15606==
==15606==
==15606== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==15606== malloc/free: in use at exit: 8 bytes in 2 blocks.
==15606== malloc/free: 4 allocs, 2 frees, 16 bytes allocated.
==15606== For counts of detected errors, rerun with: -v
==15606== searching for pointers to 2 not-freed blocks.
==15606== checked 4455600 bytes.
==15606==
==15606== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==15606==   at 0x4002978F: __builtin_new (vg_replace_malloc.c:172)
==15606==   by 0x400297E6: operator new(unsigned) (vg_replace_malloc.c:185)
==15606==   by 0x804875B: Simple::Simple() (leaky.cpp:8)
==15606==   by 0x8048648: main (leaky.cpp:24)
==15606==
==15606== 4 bytes in 1 blocks are definitely lost in loss record 2 of 2
==15606==   at 0x4002978F: __builtin_new (vg_replace_malloc.c:172)
==15606==   by 0x400297E6: operator new(unsigned) (vg_replace_malloc.c:185)
==15606==   by 0x8048633: main (leaky.cpp:24)
==15606==   by 0x4031FA46: __libc_start_main (in /lib/libc-2.3.2.so)
==15606==
==15606== LEAK SUMMARY:
==15606==   definitely lost: 4 bytes in 1 blocks.
==15606==   possibly lost:   0 bytes in 0 blocks.
==15606==   still reachable: 4 bytes in 1 blocks.
==15606==   suppressed: 0 bytes in 0 blocks.
```

当然，像 `valgrind` 这样的程序并不能具体修复内存泄漏——那还有什么意思啊？你可以使用这些工

具提供的信息找到实际的问题。一般情况下，它会一步步查找代码来找出在什么地方重写了指向对象的指针但是没有释放原始的对象。一些调试器提供了“监视点”功能，它能在发生内存泄漏的地方停止执行程序。

### 智能指针

要避免内存泄漏，有一种日益流行的技术，这就是使用智能指针（smart pointer）。智能指针概念源于这样一个事实，即如果把一切都放在栈中，这就可以避免与内存相关的大多数问题。栈比堆更安全，因为栈变量超出作用域时，它们会自动撤销和清除。智能指针结合了栈变量的安全性和堆变量的灵活性。

智能指针的基本理论十分简单，它是一个带有关联指针的对象。当智能指针超出作用域时，会删除关联的指针。本质上讲，就是在一个基于栈的对象内部包装一个堆对象。

C++ 标准模板库包含了一个智能指针的基本实现，叫做 `auto_ptr`。可以把动态分配的对象存储在基于栈的 `auto_ptr` 实例中，而不是存储在指针中。不需要显式地释放与 `auto_ptr` 关联的内存——`auto_ptr` 超出作用域时，与之关联的内存会得到清除。

作为一个例子，请考虑下面这个函数，它在堆上分配了一个 `Simple` 对象，但是忘记将其释放，这明显存在内存泄漏。

```
void leaky()
{
    Simple* mySimplePtr = new Simple(); // BUG! Memory is never released!

    mySimplePtr->go();
}
```

通过使用 `auto_ptr` 类，仍然没有显式地删除对象。然而，当 `auto_ptr` 对象超出作用域时（方法的最后），它会在析构函数中释放这个 `Simple` 对象。

```
void notLeaky()
{
    auto_ptr<Simple> mySimpleSmartPtr(new Simple());

    mySimpleSmartPtr->go();
}
```

智能指针一个最大的特征是，它们在不要用户学习大量新语法的前提下提供了众多好处。在前面这段代码中可以看到，智能指针也可以像标准指针一样解除引用（使用 `*` 或 `->`）。

第 16 章通过操作符重载提供了一个智能指针模板类的具体实现。第 25 章介绍了智能指针的一个改进实现，其中包含了引用计数。

### 13.5.3 二次删除与无效指针

一旦使用 `delete` 释放了与指针关联的内存，程序中的其他部分就可以使用这段内存了。但是，没有什么能阻止你试图继续使用这个指针。二次删除也是一个问题。如果在指针上第二次使用 `delete`，程序可能会释放已经指派给另一个对象的内存。

二次删除和使用已经释放的内存都是很难处理的问题，因为它的症状不会立即暴露出来。如果在相对较短的时间内发生了两次删除操作，程序可能会正常地运行下去，因为关联的内存没有这么快就得到重新使用。类似地，如果一个对象在删除之后立即又得到使用，那么所使用的对象极有可能还是原来的对象（而不是新指派的对象）。

当然，并不能保证总有这样的行为，也不能保证一定会继续运行。一旦对象被删除，内存分配器就



没有义务再保留它了。即使确实还能使用这个已经删除的对象，但这也是一种相当不好的程序设计风格。

许多内存泄漏检查程序（比如 valgrind）也会检查二次删除和使用已释放的对象等问题。

#### 13.5.4 访问越界指针

这一章前面曾经指出，由于指针只是内存地址，所以可能有指向内存中随机位置的指针。很容易落入这种情况。例如，请考虑一个 C 风格的字符串，由于某种原因它丢掉了结束字符 ‘\0’。而下面这个函数试图把整个字符串都用字符 ‘m’ 填充，但实际上，它还会把该字符串后面的内存也用 ‘m’ 填充。

```
void fillWithM(char* inStr)
{
    int i = 0;

    while (inStr[i] != '\0') {
        inStr[i] = 'm';
        i++;
    }
}
```

如果把未能正确终止的字符串传给这个函数，那么肯定会重写内存中某个重要的基本部分，而且程序崩溃也只是早晚的事情。考虑一下，如果与程序中对象关联的内存突然用 ‘m’ 重写了会怎么样。估计不是什么好事情！

导致越界（超过数组）写内存的 bug 经常称为缓冲区溢出错误（buffer overflow error）。这样的 bug 已经被一些能力极强的病毒和蠕虫所利用。狡猾的黑客也可以利用这一点重写内存的某一部分，从而向以上正在运行的程序注入代码。

幸运的是，许多内存检测工具也可以检测出缓冲区溢出。而且，尽管向 C 风格的字符串和数组写入内容时存在大量相关的 bug，但是如果使用像 C++ 字符串和向量这样一些高级构造的话，将有助于防止这样一些 bug。

### 13.6 小结

本章介绍了动态内存，不仅指出了基本语法，还涉及到有关的底层内容。除了使用内存检测工具和谨慎地编写代码之外，要避免出现与动态内存相关的问题，还有关键的两点。第一，需要理解指针在后台是如何工作的。在阅读指针的两个不同思维模型时，希望你确实知道编译器如何分配内存。第二，通过用基于栈的对象来“包装”指针，比如 C++ 字符串类和智能指针，就可以避免各种动态内存问题。

## 第 14 章 揭开 C++ I/O 的神秘面纱

程序的主要工作是接受输入、产生输出。不产生任何输出的程序是没有什么用处的。所有的程序设计语言都提供了某种 I/O 机制，它或者是语言的内置部分，或者是通过具体的 OS 钩子提供的。一个好的 I/O 系统应该很灵活，而且易于使用。要做到灵活，就是要提供多态性，灵活的 I/O 系统支持通过多种设备的输入输出，比如文件和用户控制台。它们还支持对不同类型的数据进行读写。I/O 很容易导致错误，因为来自用户的数据很可能是不正确的，也有可能底层的文件系统或其他数据源不可访问。因此，好的 I/O 系统还应该能够处理错误情况。

如果你熟悉 C 语言，肯定用过 `printf()` 和 `scanf()`。作为一种 I/O 机制，`printf()` 和 `scanf()` 的确是灵活的。通过转义代码和变量占位符，可以对 `printf()` 和 `scanf()` 进行定制来读入特定格式的数据，或者把一个整数输出为一个字符串。但是，衡量什么是好的 I/O 系统时还有其他一些准则，在这样一些准则面前，`printf()` 和 `scanf()` 就有点步伐踉跄了。`printf()` 和 `scanf()` 不能很好地处理错误，也没有足够的灵活性来处理定制的数据类型，最糟糕的是，在面向对象语言中，比如 C++，`printf()` 和 `scanf()` 根本就不是面向对象的！

C++ 通过流（stream）机制提供了更先进的输入输出方法。流是一种灵活的面向对象 I/O 方法。你将在本章中学到如何把流用于数据输入输出。还将学到如何使用流从各种源中读取数据，如何向各种目的写入数据，比如用户控制台、文件甚至字符串等等。本章涵盖了平常使用最多的 I/O 特性。同时还介绍了一个日益重要的话题，即如何编写可以本地化从而在世界不同区域使用的程序。

### 14.1 使用流

要习惯流这种隐喻说法还是需要花一点功夫的。最初，流看起来好像比传统的 C 风格的 I/O 更复杂，比如 `printf()`。实际上，流乍看起来比较复杂，只是因为流背后还有比 `printf()` 更深的含义。但是不要担心，通过几个例子的介绍，就能驾轻就熟了。

#### 14.1.1 到底什么是流

在第 1 章已经读到，对于数据，`cout` 流就像洗衣房的清洗槽一样。把某些变量丢进流中，它们就会写到用户屏幕或者控制台（console）上。更一般的观点是，所有的流都可以看作是数据的清洗槽。不同的流只是方向以及与之关联的源和目的有所不同。比如，你所熟悉的 `cout` 流是一个输出流，所以它的方向是“出”。它把数据写到控制台上，所以它关联的目的是“控制台”。还有一种标准的流叫做 `cin`，它从用户接收输入。`cin` 的方向是入，相关联的源是控制台。`cout` 和 `cin` 都是在 C++ 的 `std` 命名空间中预定义的流实例。

每个输入流都有一个相关联的源。每个输出流都有一个相关联的目的。

#### 14.1.2 流的源与目的

流概念可以应用于接受数据或者发出数据的任何对象。可以编写一个基于流的网络类，或者基于流

来访问基于 MIDI 的设备。在 C++ 中，流有三种常见的源和目的。

你已经阅读了用户流或控制台流的许多例子。利用控制台输入流，允许用户在运行时提供输入，从而建立交互的程序。控制台输出流则会向用户提供反馈，并输出结果。

文件流，顾名思义，就是从文件系统中读取数据或者向文件系统写入数据。文件输入流对于读入配置数据和以前保存的文件很有用，另外还有助于基于文件的数据批处理。文件输出流可用于保存状态和提供输出。

字符串流是流隐喻在 string 类型上的应用。利用字符串流，就可以把字符数据像其他的任何流一样处理了。在大部分情况下，这只是一种便利的语法，有关功能完全可以通过 string 类的方法来处理。但是，使用流语法提供了一个优化的机会，它比直接使用 string 类要方便得多。

这一节余下的部分主要介绍控制台流（cin 和 cout）。本章的后面会介绍文件流和字符串流的例子。其他类型的流，比如打印机输出或者网络 I/O 是由操作系统提供的，并没有内置到 C++ 语言中。

### 14.1.3 流输出

使用流进行输出已经在第 1 章介绍过，并且几乎本书的每一章中都使用了流来输出。本小节将简单回顾一下这些基本知识，然后介绍更高级的技术。

#### 输出基础

输出流是在头文件 <ostream> 中定义的。大部分程序员都会在他们的程序中包含 <iostream>，这样就包含了输入流和输出流的头文件。头文件 <iostream> 还声明了标准控制台输出流 cout。

<< 操作符是使用输出流的最简单的方法。C++ 的基本类型，比如 int、指针、double、字符等等，都可以使用 << 输出。此外，C++ 的 string 类也与 << 兼容，C 风格的字符串也能正确地得到输出。下面是使用 << 的几个例子，在此还提供了各个例子相应的输出。

```
int i = 7;
cout << i;
```

7

```
char ch = 'a';
cout << ch;
```

a

```
string myString = "Marni is adorable.";
cout << myString;
```

Marni is adorable.

cout 流是把数据写到控制台或者标准输出（standard output）的内置流。回想一下，可以把 << 串起来使用，从而输出多个数据。这是因为，<< 操作符返回流作为其结果，所以可以直接在同一个流上再次使用 <<。比如：

```
int i = 11;
cout << "On a scale of 1 to cute, Marni ranks " << i << "!\n";
```

On a scale of 1 to cute, Marni ranks 11!

C++ 流可以正确地解析 C 风格的转义代码，比如包含 \n 的字符串，但是在这种情况下使用内置的 endl 机制会更好。下面这个例子使用了 endl，它是在命名空间 std 中定义的，用于表示行结束字符并刷新

输出缓冲区。使用一行代码可以输出几行文本。

```
cout << "Line 1" << endl << "Line 2" << endl << "Line 3" << endl;
```

```
Line 1  
Line 2  
Line 3
```

### 输出流方法

毫无疑问，操作符<<是输出流中最有用的部分。但是，还有额外的一个功能需要研究一下。如果看一下头文件<ostream>，就会发现操作符<<有许多重载定义。还会发现一些有用的公共方法。

#### put() 和 write()

put()和 write()是原始的输出方法 (raw output method)。这两个方法分别接受一个字符或字符数组，而不是取已经定义了某种输出行为的对象或者变量。传递给这两个方法的数据按照原来的格式输出，没有进行任何专门的格式化或者处理。转义字符（比如 \n）仍然按照正确的形式输出（也就是一个回车），但是不会发生多态输出。下面这个函数采用一个 C 风格的字符串作为参数，并把它输出到控制台，这里没有使用<<操作符：

```
void rawWrite(const char* data, int dataSize)  
{  
    cout.write(data, dataSize);  
}
```

接下来这个函数使用 put 方法按规定的索引把一个 C 风格字符串的一个字符写到控制台。

```
void rawPutChar(const char* data, int charIndex)  
{  
    cout.put(data[charIndex]);  
}
```

#### flush()

向输出流写数据时，流不必立刻把数据写到目的中。大部分的输出流都会进入缓冲区 (buffer)，或者累积数据，而不是数据一到来就写出。当下列条件之一发生时，流会刷新输出 (flush)，或者写出累积的数据：

- 到达一个标记，比如 endl 标记。
- 流超出了作用域，因此被撤销。
- 对应的输入流请求输入（也就是说，利用 cin 输入时，cout 会刷新输出）。在 14.3 节中，你会学到如何建立此类链接。
- 流缓冲区已满。
- 明确告诉流要刷新输出其缓冲区。

明确告诉流进行刷新输出的一种方法是调用方法 flush()，如下面这段代码所示。

```
cout << "abc";  
cout.flush();    // abc is written to the console.  
cout << "def";  
cout << endl;    // def is written to the console.
```

不是所有的输出流都会缓冲。比如，`cerr` 流就没有对其输出进行缓冲。

### 处理输出错误

输出错误可能会在各种情况下产生。可能你试图向不存在或者已经指定为只读的文件写入数据。可能一个磁盘错误阻止了写操作成功执行，或者由于某种原因控制台进入了锁定状态。为简洁起见，到目前为止读到的所有关于流的代码都没有考虑这些可能性。但是在专业的 C++ 程序中，考虑可能发生的所有错误条件是必需的。

流处于正常的可用状态时，我们说流是“好的”。可以在流上直接调用方法 `good()` 来确定流当前是否是好的。

```
if (cout.good()) {  
    cout << "All good" << endl;  
}
```

使用方法 `good()` 能够很容易地获取关于流有效性的基本信息，但是它不能告诉你为什么流是不可用的。还有一个方法 `bad()`，它提供了更多的信息。如果 `bad()` 返回 `true`，表示发生了一个致命的错误（不同于诸如文件结束等非致命条件）。还有另一个方法 `fail()`，如果最近的操作失败，它返回 `true`，意味着下一个操作也会失败。比如，在输出流上调用 `flush()` 之后，可以调用 `fail()` 来确定流是否仍然是可用的。

```
cout.flush();  
if (cout.fail()) {  
    cerr << "Unable to flush to standard out" << endl;  
}
```

要重置流的错误状态，可以使用 `clear()` 方法：

```
cout.clear();
```

相对于文件输出流或输入流，为控制台输出流完成的错误检查要少一些。这里讨论的方法也可以应用于其他类型的流，下面在讨论其他各种类型的流时还会再次谈到这些方法。

### 输出控制符

流有许多不寻常的特性，其中之一就是，向清洗槽中抛入的可以不仅限于数据。C++ 的流还可以识别出控制符（manipulator），这些控制符是一些对象，它们可以改变流的行为，而不是为流提供数据，或者除了为流提供数据之外还可以改变流的行为。

你已经看到过一个控制符 `endl`。`endl` 控制符封装了数据和行为。它告诉流输出一个回车并刷新输出其缓冲区。下面是一些其他有用的控制符，其中许多都是在标准头文件 `<ios>` 和 `<iomanip>` 中定义的。

- `boolalpha` 和 `noboolalpha`。告诉流把 `bool` 输出为 `true` 和 `false`（`boolalpha`）或者 `1` 和 `0`（`noboolalpha`）。默认为 `noboolalpha`。
- `hex`、`oct` 和 `dec`。分别以十六进制、八进制和十进制输出数字。
- `setprecision`。设置输出小数时的小数位数（小数点后有几位）。这是一个参数化的控制符（也就是说它可以取一个参数）。
- `setw`。设置输出数值数据时的字段宽度。这是一个参数化的控制符。
- `setfill`。数字宽度小于指定宽度时，这个控制符可以指定填充空位的字符。这是一个参数化的控

制符。

- Showpoint 和 noshowpoint。对于没有小数部分的浮点数或者双精度数，这个控制符强制流总是 (Showpoint) 或者 (noshowpoint) 从不显示小数点。

下面这段程序使用了其中的几个控制符来定制输出。

```
#include <iostream>
#include <iomanip>

using namespace std;
int main(int argc, char** argv)
{
    bool myBool = true;
    cout << "This should be true: " << boolalpha << myBool << endl;
    cout << "This should be 1: " << noboolalpha << myBool << endl;
    double dbl = 1.452;
    cout << "This should be @01.452: " << setw(7) << setfill('@') << dbl << endl;
}
```

如果对控制符的概念不感兴趣，也完全可以不用它们。流通过一些与之等效的方法，比如 `setPrecision()`，也可以提供相同的功能。详细内容请参阅附录 B。

#### 14.1.4 流输入

输入流提供了一种读取结构化或者非结构化数据的简单方法。在本节中，我们将讨论 `cin`（控制台输入流）的有关输入技术。

##### 输入基础

使用输入流读取数据有两种很容易的方法。第一种方法是使用一个读取数据的操作符，对应于向输出流输出数据的 `<<` 操作符，读取数据的相应操作符 `>>`。使用 `>>` 从输入流读取数据时，提供的变量用于存储收到的值。比如，下面这段程序从用户读入一行输入，并把它放到一个字符串中。然后把字符串输出到控制台。

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char** argv)
{
    string userInput;
    cin >> userInput;
    cout << "User input was " << userInput << endl;
}
```

默认地，输入流会根据空白符对值进行词法分析 (tokenize)。比如，如果用户运行上面这段程序，并键入 `hello there` 作为输入，那么只有到第一个空白符（这个例子中的空白符是空格字符）之前的字符可以捕获到变量 `userInput` 中。输出结果如下：

```
User input was hello
```

操作符 `>>` 可以对不同的变量类型进行处理，就像操作符 `<<` 一样。比如，要读取一个整数，代码只是在变量类型上有所区别：



```
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    int userInput;
    cin >> userInput;
    cout << "User input was " << userInput << endl;
}
```

可以使用输入流读取多个值，必要时可以混合和匹配不同的类型。比如，下面这个函数是从一个饭店预定系统中摘出来的，它要询问用户的姓氏和参加宴会的人数。

```
void getReservationData()
{
    string guestName;
    int partySize;

    cout << "Name and number of guests: ";
    cin >> guestName >> partySize;
    cout << "Thank you, " << guestName << ", " << endl;
    if (partySize > 10) {
        cout << "An extra gratuity will apply." << endl;
    }
}
```

注意，即使使用 `cout` 时没有使用 `endl` 或 `flush()` 明确刷新输出缓冲区，文本仍然会写到控制台上，因为使用 `cin` 就会立刻刷新输入 `cout` 缓冲区。`cin` 和 `cout` 就是以这种方式链接在一起的。

如果混淆了 `<<` 和 `>>`，只要想一想，这两个操作符的尖角正是指向其目的。在输入流中，`<<` 指向流本身，因为数据是发送给流的。在输出流中，`>>` 指向变量，因为数据要存储到变量中。

### 输入方法

就像输出流一样，输入流也有一些方法可以用来完成更底层的访问，这些是更通用的操作符 `>>` 无法提供的功能。

#### `get()`

`get()` 方法允许从流输入原始数据。最简单的 `get()` 方法仅仅返回流中的下一个字符，但是还有其他的 `get()` 方法可以一次读取多个字符。`()` 经常用于避免使用 `>>` 操作符时发生的自动词法分析。比如，下面这个函数从输入流读取一个名字（这个名字可能由几个单词组成），直到到达流的结尾为止。

```
string readName(istream& inStream)
{
    string name;
    while (inStream.good()) {
        int next = inStream.get();
        if (next == EOF) break;
        name += next; // Implicitly convert to a char and append.
    }

    return name;
}
```

可以从几个有趣的角度来理解这个函数。首先，函数的参数是 `istream` 的引用，而不是 `const` 引用。

从流中读取数据的方法会改变所读取的具体流（最值得注意的是，会改变流的位置），所以它们不是 const 方法。因此，不能对 const 引用调用这些方法。第二，get() 的返回值是存储在 int 中，而不是 char。因为 get() 可以返回特殊的非字符值，比如 EOF（文件结束），所以这里使用了 int。把 next 追加到 string 时，会把它隐式地转换为 char。

前面这个函数看起来有点奇怪，因为可以有两种方法跳出循环。流进入“不好”状态时会跳出循环，或者是到达流的结尾时跳出循环。要从流读取数据，更常用的模式是使用另一个版本的 get() 方法，它取字符的引用作为参数，并返回流的引用。这种模式利用了这样一个事实：在有条件的上下文中计算输入流时，只有当流还可以做进一步读取时才能得到 true 结果。遇到错误或者到达文件的结尾都会导致流计算失败。实现这个特性所需转换操作的底层细节将在第 16 章中解释。下面是这个函数的另一个版本，这个版本更简洁一些。

```
string readName(istream& inStream)
{
    string name;
    char next;    while (inStream.get(next)) {
        name += next;
    }

    return name;
}

unget()
```

对于大部分目的而言，理解输入流的正确方法是把它看作是单向的清洗槽。数据会落入清洗槽并进入变量中。方法 unget() 则以某种方式打破了这个模型，它允许把数据推回到清洗槽中。

一次 unget() 调用会引起流后退一个位置，其本质是把最后一个字符读回到流中。

```
char ch1, ch2, ch3;

in >> ch1 >> ch2 >> ch3;
in.unget();
char testChar;
in >> testChar;

// testChar == ch3 at this point
```

putback()

putback() 就像 unget() 一样，允许在输入流中后退一个字符。二者的区别是，putback() 方法取流中要后退的字符作为参数：

```
char ch1;

in >> ch1;
in.putback(ch1);

// ch1 will be the next character read off the stream.
```

peek()

peek() 方法允许预览下一个值，即若是调用 get() 方法的话会返回什么值。再把清洗槽的这个比方延伸一点，可以把这个方法看作是查找清洗槽，但是并没有值真正落入清洗槽中。

在读取值之前需要向前看时，对于这样一些情况，`peek()`再合适不过了。比如，程序可能要根据下一个值是否以数字开头来决定做什么事情，如下面这个代码段所示。

```
int next = cin.peek();
if (isdigit(next)) {
    processNumber();
} else {
    processText();
}
```

`getline()`

从输入流获取一行数据太常见了，所以专门存在这样一个方法来完成这项工作。方法 `getline()` 使用一行不超过指定长度的数据来填充一个字符缓冲区，如下面这两行代码所示。

```
char buffer[kBufferSize + 1];
cin.getline(buffer, kBufferSize);
```

注意，方法 `getline()` 会从流中删除换行字符。所得到的字符串不包括换行字符。还有一种形式的 `get()` 方法可以完成 `getline()` 同样的操作，只是它会把换行字符留在输入流中。

还有一个函数 `getline()` 可以用于 C++ 字符串。它是在 `std` 命名空间中定义的，取一个流引用、一个 `string` 引用和一个可选的分隔符作为参数：

```
string myString;
std::getline(cin, myString);
```

### 处理输入错误

输入流提供了大量方法用来检测非正常的情况。与输入流相关的大部分错误条件都是在没有数据可读时产生的。比如，可能是到了流的结尾（也称为文件结尾（end of file），甚至对于非文件流也可以称之为文件结尾）。要查询输入流的状态，最常用的方法是在一个有条件的上下文中访问输入流，如上所述。也可以像输出流一样调用 `good()` 方法。还有一个方法 `eof()`，如果已经到了输入流结尾，这个方法会返回 `true`。

你应该已经习惯于在读取数据之后检查输入流的状态，这样在遭遇不好的输入时就可以恢复了。

下面这段程序使用了从流中读取数据和处理错误的常用模式。它从标准输入读取数字，一旦到了文件结尾就显示所有数字的累加和。注意，在大部分命令行环境中，文件结尾是在控制台上用 `control-D` 指示的。

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    int sum = 0;

    if (!cin.good()) {
        cout << "Standard input is in a bad state!" << endl;
        exit(1);
    }
}
```

```
int number;
while (true) {
    cin >> number;
    if (cin.good()) {
        sum += number;
    } else if (cin.eof()) {
        break; // Reached end of file
    } else {
        // Error!
        cin.clear(); // Clear the error state.
        string badToken;
        cin >> badToken; // Consume the bad input.
        cerr << "WARNING: Bad input encountered: " << badToken << endl;
    }
}

cout << "The sum is " << sum << endl;

return 0;
}
```

### 输入控制符

下面这个列表中描述的内置输入控制符可以发送到输入流，用于定制读取数据的方式。

- `boolalpha` 和 `noboolalpha`。如果使用 `boolalpha`，字符串 `true` 会解释为布尔值的 `true`，`false` 会作为布尔值的 `false` 处理。如果设置了 `noboolalpha`，则不会如此。默认的是 `noboolalpha`。
- `hex`、`oct` 和 `dec`。分别以十六进制、八进制和十进制读取数字。
- `skipws` 和 `noskipws`。告诉流在做词法分析时忽略空白符，或者把空白符读入作为一个空白符 `token`。
- `ws`。这是一个便利控制符，它只是忽略流中当前位置上的一串空白。

#### 14.1.5 输入与输出对象

前面已经看到，可以使用操作符 `<<` 来输出 C++ 字符串，尽管字符串并不是基本类型。在 C++ 中，对象可以预定如何进行输入输出。这是通过重载（overloading）操作符 `<<`，使之了解如何输出一个新类型或类来做到的。

为什么要重载 `<<`？如果熟悉 C 中的 `printf()` 函数，就会知道，在这个领域中，`printf()` 并不灵活。`printf()` 只知道几种数据类型，而且没有什么方法可以告诉这个函数另外的知识。比如，考虑以下这个简单的类：

```
class Muffin
{
public:
    string    getDescription() const;
    void      setDescription(const string& inDescription);
    int       getSize() const;
    void      setSize(int inSize);
    bool      getHasChocolateChips() const;
    void      setHasChocolateChips(bool inChips);

protected:
```

```

        string    mDescription;
        int       mSize;
        bool      mHasChocolateChips;
    };

    string Muffin::getDescription() const { return mDescription; }
    void Muffin::setDescription(const string& inDescription)
    {
        mDescription = inDescription;
    }

    int Muffin::getSize() const { return mSize; }
    void Muffin::setSize(int inSize) { mSize = inSize; }
    bool Muffin::getHasChocolateChips() const { return mHasChocolateChips; }
    void Muffin::setHasChocolateChips(bool inChips) { mHasChocolateChips = inChips; }

```

要使用 `printf()` 输出 Muffin 类的一个对象，要是能简单地把它指定为一个参数（比如使用 `%m` 作为占位符）就好了：

```
printf("Muffin output: %m\n", myMuffin); // BUG! printf doesn't understand Muffin.
```

遗憾的是，函数 `printf()` 不知道关于 Muffin 类型的任何信息，它也无法输出 Muffin 类型的对象。更糟糕的是，根据声明 `printf()` 函数所采用的方式，这会导致运行时错误，而不是编译时错误（不过，一个好的编译器会发出警告）。

所能做的最多只是使用 `printf()` 为类 Muffin 增加一个新的 `output()` 方法。

```

class Muffin
{
public:
    string    getDescription() const;
    void      setDescription(const string& inDescription);
    int       getSize() const;
    void      setSize(int inSize);
    bool      getHasChocolateChips() const;
    void      setHasChocolateChips(bool inChips);

    void output();

protected:
    string    mDescription;
    int       mSize;
    bool      mHasChocolateChips;
};

string Muffin::getDescription() const { return mDescription; }
void Muffin::setDescription(const string& inDescription) { mDescription = inDescription; }
int Muffin::getSize() const { return mSize; }
void Muffin::setSize(int inSize) { mSize = inSize; }
bool Muffin::getHasChocolateChips() const { return mHasChocolateChips; }
void Muffin::setHasChocolateChips(bool inChips) { mHasChocolateChips = inChips; }

void Muffin::output()
{
    printf("%s, Size is %d, %s\n", getDescription().c_str(), getSize(),
        (getHasChocolateChips() ? "has chips" : "no chips"));
}

```



但是使用这样一种机制是很麻烦的。要在另一行文本的中间输出 Muffin，需要把这一行分成两行调用，并在中间调用 Muffin::output()，如下所示：

```
printf("The muffin is ");  
myMuffin.output();  
printf(" -- yummy!\n");
```

通过重载<<操作符，就可以像输出字符串一样输出 Muffin——只要把它提供为<<的一个实参即可。第 16 章将详细介绍如何重载<<和>>操作符。

## 14.2 字符串流

字符串流提供了一种对 string 应用流语义的方法。采用这种方法，可以有一个表示文本数据的内存中流 (inmemory stream)。如果多个线程都向同一个字符串提供数据，或者需要把一个 string 传递给不同的函数，而同时还要维护当前的读取位置，在这样一些应用中，这种方法就很有用。因为流具有内置的词法分析功能，所以字符串流对于解析文本也很有用处。

类 ostringstream 和 istream 分别用于向字符串写数据和从字符串读数据。它们都是在头文件 <sstream> 中定义的。因为 ostringstream 和 istream 继承了 ostream 和 istream 同样的行为，所以使用起来是很类似的。

下面这段简单的程序请求用户提供单词，并把这些单词输出到一个 ostringstream 中，各单词之间用 tab 字符分开。在程序的最后，将使用 str() 方法把整个流转换成一个 string 对象，并写到控制台上。

```
#include <iostream>  
#include <sstream>  
  
using namespace std;  
int main(int argc, char** argv)  
{  
    ostringstream outStream;  
    while (cin.good()) {  
        string nextToken;  
        cout << "Next token: ";  
        cin >> nextToken;  
  
        if (nextToken == "done") break;  
        outStream << nextToken << "\t";  
    }  
  
    cout << "The end result is: " << outStream.str() << endl;  
}
```

从字符串流读取数据对我们来说也同样不陌生。下面这个函数创建一个 Muffin 对象，并从一个字符串输入流填充这个对象（看一下前面的例子）。这个流数据的格式是固定的，因此这个函数在设置对象的值时可以很容易地调用相应的 Muffin 设置函数。

```
Muffin createMuffin(istream& inStream)  
{  
    Muffin muffin;  
    // Assume data is properly formatted:  
    // Description size chips
```



```

string description;
int size;
bool hasChips;
// Read all three values. Note that chips is represented
// by the strings "true" and "false"
inStream >> description >> size >> boolalpha >> hasChips;
muffin.setSize(size);
muffin.setDescription(description);
muffin.setHasChocolateChips(hasChips);

return muffin;
}

```

把对象转变为一个“扁平”类型（比如 string）的过程通常称为编组（marshalling）。编组对于把对象存储到磁盘上或通过网络传送，将在第 24 章进一步介绍。

比起标准的 C++ 字符串来说，字符串流的主要优点是，除数据之外，对象知道自己的当前位置。取决于字符串流的特定实现，还可能其他的性能改善。

### 14.3 文件流

文件非常适合于流抽象，因为除数据之外，读写文件总是会涉及到位置。在 C++ 中，类 ofstream 和 ifstream 提供了文件的输入输出功能。这两个类在头文件 <fstream> 中定义。

处理文件系统时，检测和处理错误情况特别重要。正在操作的文件有可能存储在刚刚离线的一个网络文件存储器上。你可能想把数据写入一个文件，但是当前用户没有编辑这个文件的权限。使用上面介绍的标准错误处理机制可以检测到这些条件。

输出文件流与其他输出流之间惟一的主要区别是，文件流构造函数取文件名和你想以何种模式打开该文件作为参数。默认的模式是写，这会在文件开始处写入数据，而覆盖现有的数据。可以使用常量 ios\_base::app 以追加的模式打开输出文件流。

下面这段简单的程序会打开文件 test，并把参数输出给程序。

```

#include <iostream>
#include <fstream>

using namespace std;
int main(int argc, char** argv)
{
    ofstream outFile("test");
    if (!outFile.good()) {
        cerr << "Error while opening output file!" << endl;
        return -1;
    }
    outFile << "There were " << argc << " arguments to this program." << endl;
    outFile << "They are: " << endl;
    for (int i = 0; i < argc; i++) {
        outFile << argv[i] << endl;
    }
}

```

### 14.3.1 使用 seek() 与 tell()

所有的输入和输出流都有 seek() 和 tell() 方法, 但是在文件流上下文以外, 它们几乎没有什么意义。

使用方法 seek() 可以移动到输入或输出流的任意位置。这样移动会打破流所具有的隐喻含义, 所以最好少用这些方法。seek() 有几种形式。输入流中的 seek() 方法实际上名为 seekg() (g 指 get), 输出流中的 seek() 方法名为 seekp() (p 指 put)。每种类型的流都有两个查找方法。你可以在流中查找绝对位置, 比如开头或者第 17 个位置, 或者查找某个相对位置 (指定偏移量), 比如从当前标志起的第 3 个位置。位置以字节为单位度量。

要在输出流中查找绝对位置, 可以使用带一个参数的 seekp() 方法, 像下例所示, 它使用常量 ios\_base::beg 移动到流的开始位置。还有用于移动到流结尾的常量 (ios\_base::end) 和移动到流中间的常量 (ios\_base::cur)。

```
ostream.seekp(ios_base::beg);
```

在输入流中查找绝对位置也是一样的, 只是使用的是 seekg() 方法:

```
istream.seekg(ios_base::beg);
```

带两个参数的 seek() 方法会移动到流中的一个相对位置。第一个参数规定要移动多少个位置, 第二个参数说明移动的起始点。移动到相对于文件开头的位置时, 起始点参数使用常量 ios\_base::beg。移动到相对于文件结尾的位置时, 起始点参数使用 ios\_base::end。移动到相对于当前位置的位置时, 起始点参数使用 ios\_base::cur。比如, 下面这行代码是移动到相对于流的开头的第二个字符:

```
ostream.seekp(2, ios_base::beg);
```

下面这个例子是移动到输入流的倒数第三个位置。

```
istream.seekg(-3, ios_base::end);
```

也可以使用 tell() 方法查询流的当前位置。tell() 返回 ios\_base::pos\_type, 它指明流的当前位置。在完成 seek() 或查询你是否在某个特定位置之前, 可以使用这个结果记住当前标记的位置。就像使用 seek() 一样, 对于输入流和输出流, 也有几个不同的 tell() 方法。输入流使用的是 tellg(), 输出流使用的是 tellp()。

下面这行代码检查输入流的位置, 来确定它是否在开头。

```
ios_base::pos_type curPos = inStream.tellg();  
if (curPos == ios_base::beg) {  
    cout << "We're at the beginning." << endl;  
}
```

下面是一个示例程序, 它把所有这些都用上了。这个程序把数据写入一个文件 test.out, 并完成下列测试:

1. 把字符串 12345 输出到该文件。
2. 验证标志在流的位置 5 上。

3. 移到输出流中的位置 2 上。
4. 在位置 2 输出一个 0，并刷新输出流。
5. 在文件 test.out 上打开一个输入流。
6. 把第一个 token 读作为一个整数。
7. 确认该值为 12045。

```
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char** argv)
{
    ofstream fout("test.out");
    if (!fout) {
        cerr << "Error opening test.out for writing\n";
        exit(1);
    }

    // 1. Output the string "12345".
    fout << "12345";

    // 2. Verify that the marker is at the end.
    ios_base::pos_type curPos = fout.tellp();    if (curPos == 5) {
        cout << "Test passed: Currently at position 5" << endl;
    } else {
        cout << "Test failed: Not at position 5" << endl;
    }

    // 3. Move to position 2 in the stream.
    fout.seekp(2, ios_base::beg);

    // 4. Output a 0 in position 2 and flush the stream.
    fout << 0;
    fout.flush();

    // 5. Open an input stream on test.out.
    ifstream fin("test.out");
    if (!fin) {
        cerr << "Error opening test.out for reading\n";
        exit(1);
    }

    // 6. Read the first token as an integer.
    int testVal;
    fin >> testVal;

    // 7. Confirm that the value is 12045.
    if (testVal == 12045) {
        cout << "Test passed: Value is 12045" << endl;
    } else {
        cout << "Test failed: Value is not 12045";
    }
}
```

### 14.3.2 链接流

可以在任何输入流与输出流之间建立链接，从而提供一种“一旦访问就刷新输出”（flush-on-access）的行为。换句话说，从输入流请求数据时，与其链接的输出流会自动刷新输出。这个行为可以用于所有的流，但是对于可能相互依赖的文件流尤其有用。

流链接用方法 `tie()` 来实现。要把输出流绑定到一个输入流上，可以在输入流上调用 `tie()`，并把输出流的地址传递给它。要断开这个链接，传递 `NULL` 即可。

下面这段程序把一个文件的输入流绑到了另一个文件的输出流上。也可以把它绑到同一个文件的输出流上，不过要同时读写同一个文件，双向 I/O（下面会介绍）可能更好。

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(int argc, char** argv)
{
    ifstream inFile("input.txt");
    ofstream outFile("output.txt");

    // Set up a link between inFile and outFile.
    inFile.tie(&outFile);

    // Output some text to outFile. Normally, this would
    // not flush because std::endl was not sent.
    outFile << "Hello there!";

    // outFile has NOT been flushed.

    // Read some text from inFile. This will trigger flush()
    // on outFile.
    string nextToken;
    inFile >> nextToken;

    // outFile HAS been flushed.
}
```

方法 `flush()` 是在 `ostream` 基类上定义的，所以也可以把一个输出流链接到另一个输出流上。

```
outFile.tie(&anotherOutputFile);
```

这样的关系表示，每次向一个文件写数据时，就会向另一个文件写入已经发送的缓冲数据。可以使用这种机制保持两个相关文件之间的同步。

### 14.4 双向 I/O

到目前为止，本章一直是把输入输出流作为独立但相关的两个类来讨论的。实际上，还有一种流可以同时进行输入输出。双向流（bidirectional stream）可以同时作为输入流和输出流操作。

双向流是 `istream` 的子类，所以也是 `istream` 和 `ostream` 的子类，因此可以作为一个有用的多重继承的例子。如你所想，双向流同时支持 `>>` 操作符和 `<<` 操作符，还支持输入流和输出流的方法。

`fstream` 类提供了一个双向的文件流。如果应用需要替换一个文件中的数据, `fstream` 就非常理想, 因为在找到正确的位置之前可以一直完成读操作, 找到之后立即切换为写操作。比如, 考虑这样一个程序, 它存储了 ID 号和电话号码之间的一个映射列表。可能会使用以下格式的数据文件:

```
123 408-555-0394
124 415-555-3422
164 585-555-3490
100 650-555-3434
```

对于这样的程序, 一个合理的方法就是程序打开时读取整个数据文件, 程序关闭时重写该文件 (接受所做的全部修改)。但是如果数据集特别大, 可能无法把所有的数据都保存在内存中。利用 `iostream`, 就不必把所有数据保存在内存中了。可以很容易地扫描整个文件找到一条记录, 并以追加方式打开输出文件, 来增加新的记录。要修改现有的记录, 可以使用双向流, 就像下面这个函数所示, 它会修改给定 ID 对应的电话号码。

```
void changeNumberForID(const string& inFileName, int inID,
    const string& inNewNumber)
{
    fstream ioData(inFileName.c_str());
    if (!ioData) {
        cerr << "Error while opening file " << inFileName << endl;
        exit(1);
    }

    // Loop until the end of file
    while (ioData.good()) {
        int id;
        string number;

        // Read the next ID.
        ioData >> id;

        // Check to see if the current record is the one being changed.
        if (id == inID) {
            // Seek to the current read position
            ioData.seekp(ioData.tellg());

            // Output a space, then the new number.
            ioData << " " << inNewNumber;
            break;
        }

        // Read the current number to advance the stream.
        ioData >> number;
    }
}
```

当然, 只有数据长度固定时, 这样的方法才能正常工作。前面这个程序从读操作切换到写操作时, 输出数据会重写文件中的其他数据。要保留文件的当前格式, 并避免覆盖到下一条记录, 数据长度就必须相等。

通过 `string stream` 类, 也可以以双向方式访问字符串流。

双向流对于读位置和写位置分别有单独的指针。在读操作与写操作之间切换时, 需要查找到正确的位置。



## 14.5 国际化

在学习如何用 C 或 C++ 编程时，可以把字符看作与字节等价，并认为所有字符都在 ASCII (U. S.) 字符集中，这是有用的。在现实中，专业的 C++ 程序员意识到，成功的软件程序要在全世界使用。即使在编写程序时最初没有考虑到国际范围的用户，但不应该杜绝本地化 (localizing)，或者要允许软件以后能踏上国际化道路。

### 14.5.1 宽字符

把字符看作字节的一个问题就是并不是所有的编程语言或者字符集 (character set) 都能用 8 个二进制位或者 1 个字节来表示。幸运的是，C++ 有一个内置类型 `wchar_t` 可以保存宽字符 (wide character)。有非 ASCII (U. S.) 字符的语言，比如日语和阿拉伯语，在 C++ 中都可以用 `wchar_t` 表示。

如果你的程序有机会在非西方字符集环境中使用 (提示：确实有这种可能!)，就应该从一开始就使用宽字符。使用 `wchar_t` 很简单，因为它会像 `char` 一样工作。惟一的区别是字符串和字符直接量都有一个前缀字母 `L`，来指示应该使用宽字符编码。例如，要初始化 `wchar_t` 字符为字母 `m`，应该编写以下代码：

```
wchar_t myWideCharacter = L'm';
```

你喜欢的所有类型和类都有宽字符版本。宽 `string` 类就是 `wstring`。使用字母 `w` 作为前缀的模式也可以应用于流。宽字符文件输出流可以使用 `wofstream` 处理，输入流用 `wifstream` 处理。这些类名的发音 (*woof-stream?* *whiff-stream?*) 本身就很有意思，这足以让你意识到程序国际化的意义。

除了 `cout`、`cin` 和 `cerr` 外，内置的控制台和错误流也有宽版本：`wcout`、`wcin` 和 `wcerr`。就像对其他宽流类和类型一样，使用宽版本 (`wcout`、`wcin` 和 `wcerr`) 和不宽的版本 (`cout`、`cin` 和 `cerr`) 并没有什么区别，如下面这个简单程序所示：

```
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    wcout << L"I am internationally aware." << endl;
}
```

### 14.5.2 非西方字符集

宽字符使得 C++ 向前迈进了一大步，因为宽字符增加了定义单个字符的可用空间量。下一步是要明确如何使用这个空间。在传统的 (即已经过时的) ASCII 字符中，每个字母都对应于一个特定的数字。每个数字都正好保存在一个字节中，所以字母与数字是相同的，与字节也是相同的。

现代字符表示法并没有太大的区别。字符到数字的映射 (现在称为码点 (code point)) 集要大得多，因为除了讲英语的程序员熟悉的字符之外，还要处理许多不同的字符集。在已知的所有字符集中，字符到码点的映射是用 Unicode 标准定义的。比如，希伯来字符 **א** (发音为 *aleph*) 就映射到 Unicode 码点 05D0。不会再有其他任意字符集中的其他字符映射到这个码点。

要能正常使用 Unicode 文本，还要知道其编码方式 (encoding)。不同的应用程序可以用不同的方法存储 Unicode 字符。在 C++ 中，宽字符的标准编码方法称为 UTF-16，因为存储每个字符都需要 16 个二



进制位。

### 14.5.3 本地化环境与方面

各国之间表示数据的惟一区别就是字符集。即使几个国家使用类似的字符集，比如英国和美国，在如何表示类似日期和货币这样的数据上还是有区别的。

标准的 C++ 库包含了一种内置的机制，它可以把特定地方的具体数据分组到一个本地化环境 (locale) 中。本地化环境是特定位置相关设置的集合。各个设置称为一个方面 (facet)。本地化环境的一个例子就是 U. S. 英语。方面的例子是用于显示日期的格式。有几种所有本地化环境都有的内置方面。C++ 还提供了定制或者增加方面的方法。

#### 使用本地化环境

从程序员的角度来看，本地化环境是 C++ 语言的一个自动特性。使用 I/O 流时，数据要根据特定的本地化环境来完成格式化。本地化环境只不过是能够附加到流上的对象。比如，下面这行代码使用了输出流的 imbue() 方法，来把美国英语本地化环境（通常名为 “en\_U”）附加到宽字符控制台输出流上：

```
wcout.imbue(locale("en_US")); // locale is defined in the std namespace
```

美国英语一般不是默认的本地化环境。默认的本地化环境一般是经典 (classic) 本地化环境，它使用 ANSI C 的约定。经典 C 本地化环境类似于美国英语本地化环境的设置，但是稍微有一点区别。

比如，如果根本没有设置本地化环境，或者设置默认的本地化环境，并且要输出一个数字，那么输出时不带任何标点：

```
wcout.imbue(locale("C"))  
wcout << 32767 << endl;
```

这两行代码的输出结果为：

32767

但是，如果使用的本地化环境是美国英语，数字会用美国英语的标点格式化。下面这两行代码在输出数字之前把本地化环境设置为美国英语：

```
wcout.imbue(locale("en_US"));  
wcout << 32767 << endl;
```

这两行代码的输出结果为：

32,767

你可能注意到了，不同的地区有不同的方法来格式化数字数据，包括用于分隔千分位的标点和标志小数点位置的标点都可能有所不同。

可以根据具体实现来决定本地化环境的名字，不过本地化环境的大部分实现都采用了标准化做法，即语言和地区都是包括两个字母的代码，这二者之间会分隔开（译者注：可能用 “\_”，也可以用 “-”），还可以有一个可选的编码。比如，法国的法语本地化环境就是 fr\_FR。日本的用日本工业标准编码的日语本地化环境就是 ja\_JP. jis。

大多数操作系统都有一种机制来确定用户定义的本地化环境。在 C++ 中，可以向本地化环境对象构造函数传递一个空字符串，从而由用户环境创建一个本地化环境。一旦创建了这个对象，就可以用它来查询本地化环境，还可以基于这个本地化环境做一些编程方面的决策。

比如，下面这段程序创建了一个默认的本地化环境。name()方法用于得到一个描述本地化环境的 C++ 字符串。最后根据本地化环境是否是美国英语来确定输出两条消息中的哪一条。

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char** argv)
{
    locale loc("");

    if (loc.name().find("en_US") == string::npos &&
        loc.name().find("United States") == string::npos) {
        wcout << L"Welcome non-U.S. user!" << endl;
    } else {
        wcout << L"Welcome U.S. user!" << endl;
    }
}
```

根据本地化环境的名字来确定本地化环境，这不一定能够正确地确定用户是否真的在这个地方，但是可以提供一条线索。

#### 使用方面

可以使用函数 std::use\_facet() 来获取特定本地化环境中的特定方面。比如，下面这个表达式可以用来检索英国英语本地化环境的标准货币符号方面。

```
use_facet<moneypunct<wchar_t>> (locale("en_GB"));
```

注意，最内层的模板类型决定了使用哪一种字符类型，通常是 wchar\_t 或 char。使用嵌套的模板类不太好懂，但是一旦熟悉了这个语法，就可以利用这种方法得到一个对象，它包含了你需要知道的关于英国货币符号的所有信息。标准方面中可用的数据都在头文件 <locale> 及其相关文件中定义。

下面这个程序在美国英语和英国英语本地化环境中打印出货币符号，从而结合使用了本地化环境和方面。注意，取决于你的环境，英国的货币符号可能会显示为一个问号，方框或者什么也没有。还要注意，在不同的平台下本地化环境名可能有所不同。如果环境能够处理这个字符，实际上可以得到英镑符号。

```
#include <iostream>
#include <locale>

using namespace std;

int main(int argc, char** argv)
{
    locale locUSEng ("en_US");
    locale locBritishEng ("en_GB");

    wstring dollars = use_facet<moneypunct<wchar_t>> (locUSEng).curr_symbol();
    wstring pounds = use_facet<moneypunct<wchar_t>> (locBritishEng).curr_symbol();

    wcout << L"In the US, the currency symbol is " << dollars << endl;
    wcout << L"In Great Britain, the current symbol is " << pounds << endl;
}
```

## 14.6 小结

就像我们希望的，你已经发现，流提供了一种灵活、面向对象的方法来完成输入输出。这一章最重要的就是流的概念，这甚至比流的使用更重要。一些操作系统可能有自己的文件访问和 I/O 工具，不过要使用任何类型的现代 I/O 系统，了解流和类似流的库如何工作都是绝对必要的。

我们还希望你对如何编写国际化代码有所认识。有过本地化实践的人会告诉你，如果已经提前规划，使用了 Unicode 字符，并考虑到了本地化环境，那么支持一种新的语言或者本地化环境则是非常容易的。

## 第15章 处理错误

不可避免地，C++程序往往会遇到错误。程序可能无法打开一个文件，网络连接可能中断，或者用户会输入一个不正确的值，诸如此类。专业的C++程序把这些情况视作为“异常”（exceptional）而非“意外”（unexpected），并且会适当地加以处理。C++语言提供了一种称为异常（exception）的特性来支持程序中的错误处理。

到目前为止，为简单起见，本书中的代码示例基本上都忽略了错误条件。这一章将更正这种过于简化的做法，教你如何从一开始就考虑在程序中加入错误处理。本章重点介绍C++异常，包括异常语法的细节，还会介绍如何有效地驾驭异常，来创建设计合理的错误处理程序。

本章包括以下内容：

- C++错误处理概述，包括C++中异常的优点和缺点
- 异常的语法
  - 抛出和捕获异常
  - 未捕获的异常
  - 抛出列表
- 异常类层次体系和多态
  - C++异常层次体系
  - 编写自己的异常类
- 栈展开和清除
- 常见错误处理问题
  - 内存分配错误
  - 构造函数和析构函数中的错误

### 15.1 错误和异常

即使是写得很好的程序也会遇到错误和异常情况。没有程序会独立存在。它们可能离不开诸如网络 and 文件系统等外部设施，可能会依赖于第三方库等外部代码，而且还依赖于用户的输入。这些领域都可能带来异常情况。因此，无论是谁编写计算机程序，都必须在程序中包括错误处理功能。有些语言（如C）没有为错误处理提供任何特定的语言来处理。使用这些语言的程序员通常要依赖函数的返回值和其他特殊方法来处理错误。其他语言（如Java）则要求必须使用一种称为exception的语言特性作为错误处理机制。C++则介于这两个极端之间。它提供了对异常的语言支持，但是不要求一定要使用异常。不过，C++中不能完全忽略异常，因为有一些基本工具（如内存分配例程）就使用了异常。

#### 15.1.1 到底什么是异常

异常是一种机制，利用这种异常机制，一段代码可以通知另一段代码发生了一种“异常”情况或错



学习在线

视频资料下载  
电子书交流

[www.eimhe.com](http://www.eimhe.com)

误条件，而不能再沿着正常的代码路径前进。遇到错误的代码会抛出 (throw) 异常，处理异常的代码则捕获 (catch) 异常。异常并不遵循你所熟悉的基本执行规则，即不会逐步执行。当一段代码抛出一个异常时，程序会立即停止一步一步地执行代码，而把控制转移给异常处理程序 (exception handler)，异常处理程序可能就在同一个函数的下一行，也可能是栈中上面的某个函数调用。如果你喜欢拿运动打比方，可以把抛出异常的代码认为是把棒球抛回内场的外场球员，而最近的内场球员 (最近的异常处理程序) 会抓住这个球 (异常)。图 15-1 显示了包括三个函数调用的一个假想的栈。函数 A() 有异常处理程序。它调用了函数 B()，而函数 B() 又调用了函数 C()，函数 C() 则抛出了异常。

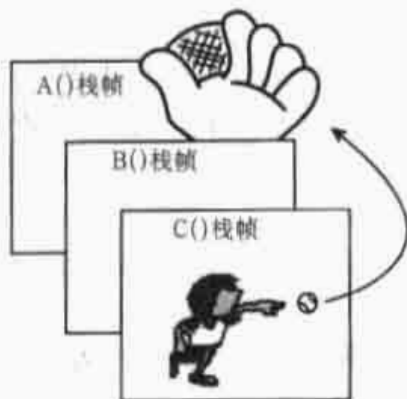


图 15-1

图 15-2 显示了捕获异常的处理程序。对应 C() 和 B() 的栈帧已经删除，只剩下 A() 的栈帧。



图 15-2

有些使用 C++ 已经多年的人了解到 C++ 还支持异常时，可能会很奇怪。程序员往往会把异常与 Java 之类的语言关联在一起，在这些语言中，异常更为明显。不过，C++ 确实对异常提供了完备的支持。

### 15.1.2 C++ 中的异常为什么好

前面已经提到，C++ 程序运行时错误是不可避免的。尽管这是事实，但是大多数 C 和 C++ 程序中的错误处理都很混乱，而且方式很特别。事实上的 C 错误处理标准就是使用整型的返回码和 `errno` 宏来通知出现了错误，而且许多 C++ 程序也沿袭了这种做法。`errno` 相当于一个全局的整型变量，函数可以用这个变量向调用函数反映错误。

遗憾的是，整型返回码和 `errno` 往往会遭到不一致的使用。有些函数返回 0 表示成功，返回 -1 表示出错。如果返回 -1，还会将 `errno` 设置为一个错误码。但其他函数可能返回 0 表示成功，返回非 0 表示出错，并用实际的返回值指定错误码。这些函数不使用 `errno`。还有一些函数可能返回 0 表示失败而不是成功，这可能是因为在 C 和 C++ 中 0 总是计算为 `false`。

由于存在这些不一致性，就会导致问题，因为程序员遇到一个新函数时，通常会认为它的返回码与其他类似函数的返回码有相同的含义。但并不一定如此。在 Solaris 9 上，就有两个不同的同步对象库：



POSIX 版本和 Solaris 版本。在 POSIX 版本中，初始化一个信号量的函数名为 `sem_init()`，而 Solaris 版本中初始化一个信号量的函数名为 `sema_init()`。这么看来好像还不算太混乱，但要知道，这两个函数会用不同的方式处理错误码。`sem_init()` 返回 -1，并根据错误设置 `errno`，而 `sema_init()` 直接将错误码作为一个正整数返回，它没有设置 `errno`。

还有一个问题，C++ 中的函数只允许有一个返回值，所以如果既想返回一个错误，又想返回一个值，就必须另寻其他方法。一种选择是通过一个引用参数返回值或错误。另外一种选择是置错误码为返回类型的一个可能值，如返回类型为指针，则可以用 NULL 指针指示错误。

异常为错误处理提供了一种更容易、更一致而且更安全的机制。相对于 C 和 C++ 中的特殊方法，异常有许多突出的优点：

- 函数的返回码可能被忽略。异常不能忽略，如果程序没有捕获一个异常，程序就会终止。
- 整型返回码未包含任何语义信息。对于不同的程序员来说，不同的数字可能有不同的含义。异常则包含语义信息，可能由类型名就可以反映出来，如果异常是对象，还可以由数据提供信息。
- 整型返回码缺乏相关的上下文信息。可以使用异常尽可能多地从发现错误的代码向处理错误的代码传递信息。异常还可以用于传递信息而不是错误，不过许多程序员都认为这是对异常机制的滥用。
- 异常处理可以在调用栈中跳级。也就是说，一个函数所处理的错误可能出现在栈中几个函数调用下的某个调用中，错误处理代码并不紧接在出错函数的后面。（译者注：这表示，在栈中，出错调用之上不一定刚好就是错误处理代码）。返回码要求各级调用栈都必须在前一级清除之后显式清除。

### 15.1.3 C++ 中的异常为什么不好

尽管一般意义上异常具有上述优点，但是 C++ 中的异常存在一些问题，以至于一些程序员对 C++ 中的异常并不认可。第一个问题是性能问题，为支持异常而增加的语言特性会使所有程序变慢，即使根本没有使用异常的程序也会因此变慢。不过，除非你在编写高性能或系统级软件，否则这也不算什么。第 17 章将更详细地讨论这个问题。

第二个问题是 C++ 中的异常支持并不是该语言中的一个集成部分，这一点不同于其他语言（如 Java）。例如，在 Java 中，如果一个函数可能抛出某些异常，但是未在可能的异常列表中指出，它就不能抛出任何异常，这是合理的。在 C++ 中，则恰恰相反，没有指定一个异常列表的函数能抛出任何想抛出的异常！另外，C++ 中并不能保证编译时异常列表，这说明函数的异常列表可能会在运行时改动。由于存在诸如此类的不一致性，使得 C++ 中的异常较难正确地使用（本不应如此），这就使一些程序员望而却步。

最后一点，异常机制会对动态分配的内存和资源清除带来问题。编程时使用异常的话，很难确保完成适当的清除。这个问题将在下面讨论。

### 15.1.4 我们的建议

尽管存在这样一些缺点，我们还是建议应当将异常作为一种有用的错误处理机制。我们认为，异常所提供的结构和错误处理形式化很有意义，相对于它不好的方面，可谓利大于弊。因此，本章余下的部分将重点介绍异常。尽管你可能不打算在程序中使用异常，也可以通读这一章，对 C++ 程序设计中一些常见的错误处理问题有所熟悉。

## 15.2 异常机制

在输入和输出领域经常会出现异常情况。以下是一个简单的函数，它要打开一个文件，从这个文件

读一个整数表，并把这些整数存储在所提供的 vector 数据结构中。

应该记得，第 4 章曾介绍过 vector 是一个动态数组。可以使用 push\_back() 方法增加元素，还可以使用数组记法来访问元素。

```
#include <fstream>
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;

    istr.open(fileName.c_str());

    // Read the integers one by one and add them to the vector.
    while (istr >> temp) {
        dest.push_back(temp);
    }
}
```

可以如下使用 readIntegerFile()：

```
int main(int argc, char** argv)
{
    vector<int> myInts;
    const string fileName = "IntegerFile.txt";

    readIntegerFile(fileName, myInts);

    for (size_t i = 0; i < myInts.size(); i++) {
        cout << myInts[i] << " ";
    }
    cout << endl;

    return (0);
}
```

你应该发现了，这些函数中缺少错误处理。这一节后面将介绍如何利用异常增加错误处理。

### 15.2.1 抛出和捕获异常

readIntegerFile() 函数中最容易出现的问题是打开文件失败。这种情况就很适合抛出一个异常。语法如下所示：

```
#include <fstream>
#include <iostream>
#include <vector>
#include <string>
#include <exception>
using namespace std;

void readIntegerFile(const string& fileName, vector<int>& dest)
```

```
{
    ifstream istr;
    int temp;

    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file; throw an exception.
        throw exception();
    }

    // Read the integers one by one and add them to the vector.
    while (istr >> temp) {
        dest.push_back(temp);
    }
}
```

throw 是 C++ 中的一个关键字，这也是抛出异常的惟一途径。C++ 提供了一个名为 exception 的类，这个类在 <exception> 头文件中声明。throw 代码行中的 exception() 部分是指要构造一个类型为 exception 的新对象以供抛出。

如果这个函数未能打开文件，而执行了 throw exception(); 这行代码就会跳过函数中余下的部分，将控制权转移给最近的异常处理程序。

如果在代码中抛出了异常，而且还编写了代码来处理这些异常，此时异常才最为有用。以下是一个 main() 函数，它会处理 readIntegerFile() 中抛出的异常：

```
int main(int argc, char** argv)
{
    vector<int> myInts;
    const string fileName = "IntegerFile.txt";

    try {
        readIntegerFile(fileName, myInts);
    } catch (const exception& e) {
        cerr << "Unable to open file " << fileName << endl;
        exit (1);
    }

    for (size_t i = 0; i < myInts.size(); i++) {
        cout << myInts[i] << " ";
    }
    cout << endl;

    return (0);
}
```

异常处理采用的做法是：先“尝试”执行一个代码块，并指定了另一个代码块对前一个代码块中可能出现的问题做出反应。在这个例子中，catch 语句对 try 块中抛出的所有 exception 类型的异常做出反应，即打印一条错误消息并退出。如果 try 块正常完成而没有抛出任何异常，就会跳过 catch 块。可以把 try/catch 块看作是“美化的”if 语句。如果在 try 块中抛出一个异常，则执行 catch 块。否则，将其跳过。

尽管默认情况下，程序流不会抛出异常，但是可以通过调用流的 exceptions() 方法告诉流为一些错误条件抛出异常。不过，就连 Bjarne Stroustrup (C++ 之父) 也不建议采用这种方法。在《The C++ Programming Language》第三版中，他指出“我更倾向于直接处理流状态。对于在函数中利用本地控制结

构来处理的工作，使用异常不会带来多少改进。”本书在这方面就采纳了他的观点。

### 15.2.2 异常类型

可以抛出任何类型的异常。前面的例子抛出了一个类型为 `exception` 的对象，不过异常不一定非得是对象。还可以如下抛出一个简单的 `int`：

```
void readIntegerFile(const string& fileName, vector<int>& dest)
{
    // Code omitted
    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw 5;
    }
    // Code omitted
}
```

然后还需要修改 `catch` 语句：

```
int main(int argc, char** argv)
{
    // code omitted
    try {
        readIntegerFile(fileName, myInts);
    } catch (int e) {
        cerr << "Unable to open file " << fileName << endl;
        exit (1);
    }

    // Code omitted
}
```

还可以抛出一个 C 风格的 `char*` 字符串。这种技术在有些时候非常有用，因为字符串可以包含有关异常的信息。

```
void readIntegerFile(const string& fileName, vector<int>& dest)
{
    // Code omitted
    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw "Unable to open file";
    }
    // Code omitted
}
```

捕获 `char*` 异常时，可以输出结果：

```
int main(int argc, char** argv)
{
    // Code omitted
    try {
        readIntegerFile(fileName, myInts);
    } catch (const char* e) {
        cerr << e << endl;
    }
}
```

```

        exit (1);
    }
    // Code omitted
}

```

不过，一般还是应该抛出对象作为异常，这有两个原因：

- 仅通过对象的类名就可以传达有关信息。
- 对象可以存储描述异常的信息，如字符串。

C++ 标准库定义了 8 个异常类，以下将做详细的介绍。你还可以编写自己的异常类。具体如何做到也将在后面详细说明。

#### 按 const 和引用捕获异常对象

在上面的例子中，readIntegerFile() 抛出一个类型为 exception 的对象，catch 行如下所示。

```

} catch (const exception& e) {

```

不过，在此没有必要按 const 引用来捕获对象。也可以按值来捕获对象，如下所示：

```

} catch (exception e) {

```

另外，也可以按引用（没有 const）捕获对象：

```

} catch (exception& e) {

```

而且，正如在 char\* 例子中所示，只要抛出了异常指针，还可以捕获异常指针。

你的程序可以按值、按引用、const 引用或指针来捕获异常。

### 15.2.3 抛出和捕获多个异常

无法打开文件并不是 readIntegerFile() 可能遇到的惟一问题。从文件读取数据时，如果格式不正确，也可能导致错误。以下是 readIntegerFile() 的一个实现，在此如果无法打开文件或者无法正确地读取数据，它都会抛出一个异常。

```

void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;

    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw exception();
    }

    // Read the integers one by one and add them to the vector.
    while (istr >> temp) {
        dest.push_back(temp);
    }

    if (istr.eof()) {
        // We reached the end-of-file.
        istr.close();
    }
}

```

```

    } else {
        // Some other error. Throw an exception.
        istr.close();
        throw exception();
    }
}

```

main() 中的代码基本上无需修改, 因为它已经捕获了一个类型为 exception 的异常。不过, 现在这个异常可能在两个不同的情况下抛出, 因此必须相应地修改错误消息:

```

int main(int argc, char** argv)
{
    // Code omitted
    try {
        readIntegerFile(fileName, myInts);
    } catch (const exception& e) {
        cerr << "Unable either to open or to read " << fileName << endl;
        exit (1);
    }
    // Code omitted
}

```

另外, 可以从 readIntegerFile() 抛出两个不同类型的异常, 这样调用者就可以区别出究竟出现了哪一种错误。以下是 readIntegerFile() 的另一个实现, 在此如果无法打开文件则抛出一个 invalid\_argument 类的异常对象, 如果无法读取整数, 则抛出一个 runtime\_exception 类的异常对象。invalid\_argument 和 runtime\_exception 类都在头文件 <stdexcept> 中定义, 它们是 C++ 标准库的一部分。

```

#include <fstream>
#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>

using namespace std;

void readIntegerFile(const string& fileName, vector<int>& dest)
{
    ifstream istr;
    int temp;

    istr.open(fileName.c_str());
    if (istr.fail()) {
        // We failed to open the file: throw an exception.
        throw invalid_argument("");
    }

    // Read the integers one by one and add them to the vector.
    while (istr >> temp) {
        dest.push_back(temp);
    }

    if (istr.eof()) {
        // We reached the end-of-file.
        istr.close();
    }
}

```



```
    } else {  
        // Some other error. Throw an exception.  
        istr.close();  
        throw runtime_error("");  
    }  
}
```

`invalid_argument` 和 `runtime_error` 没有公共的默认构造函数，只有 `string` 版本的构造函数。现在 `main()` 可以利用两个 `catch` 语句来捕获 `invalid_argument` 和 `runtime_error`：

```
int main(int argc, char** argv)  
{  
    // Code omitted  
    try {  
        readIntegerFile(fileName, myInts);  
    } catch (const invalid_argument& e) {  
        cerr << "Unable to open file " << fileName << endl;  
        exit (1);  
    } catch (const runtime_error& e) {  
        cerr << "Error reading file " << fileName << endl;  
        exit (1);  
    }  
    // Code omitted  
}
```

如果在 `try` 块中抛出了一个异常，编译器会找到与该异常类型匹配的适当的捕获处理程序。如果 `readIntegerFile()` 无法打开文件，抛出了一个 `invalid_argument` 对象，这个异常就会被第一个 `catch` 语句捕获。如果 `readIntegerFile()` 无法正确地读取文件，而抛出一个 `runtime_error`，则会由第二个 `catch` 语句来捕获这个异常。

#### 匹配和 Const

在想要捕获的异常类型前面指定 `const` 与否对于匹配来说没有任何影响。也就是说，以下这行代码可以匹配任何类型为 `runtime_error` 的异常。

```
} catch (const runtime_error& e) {
```

下面这行代码也可以匹配任何类型为 `runtime_error` 的异常：

```
} catch (runtime_error& e) {
```

一般应当在捕获异常时加上 `const`，以此表明不会修改这些异常。

#### 匹配任何异常

可以编写一行 `catch` 语句来匹配所有可能的异常，其特定语法如下所示：

```
int main(int argc, char** argv)  
{  
    // Code omitted  
    try {  
        readIntegerFile(fileName, myInts);  
    } catch (...) {  
        cerr << "Error reading or opening file " << fileName << endl;  
        exit (1);  
    }  
}
```

```

    }
    // Code omitted
}

```

上面出现的三个点并不是印刷错误。这是一个通配符，用以匹配任何异常类型。如果要调用某个代码，但该代码没有提供充分的文档，这项技术就很有用，由此可以确保捕获到所有可能的异常。不过，如果已经完全了解会抛出哪些异常，这个技术就不太好了，因为它会对每一种异常类型做相同的处理。更好的做法是明确地匹配各种异常类型，并采取适当的、有目标性的动作。

#### 15.2.4 未捕获的异常

如果你的程序抛出了一个异常，但是未在任何地方捕获，这个程序就会终止。这往往不是你的本意。异常的意义就是让程序有机会处理和修正不合适或意外的情况。如果程序没有捕获某个异常，那么开始时抛出这个异常就没有什么意义。

要捕获并处理程序中抛出的所有可能的异常。

即使不能处理某种异常，也应该编写代码来捕获这个异常，并在退出之前打印一条合适的错误消息。

如果存在一个未捕获的异常，还可以改变程序的行为（译者注：即不是简单退出）。当程序遇到一个未捕获的异常时，它会调用内置的 `terminate()` 函数，这个函数只是调用 `<cstdlib>` 中的 `abort()` 来关闭程序。可以调用 `set_terminate()`，并提供一个回调函数指针来设置自己的 `terminate_handler`，这个回调函数无参数，也不返回任何值。`terminate()`、`set_terminate()` 和 `terminate_handler` 都在 `<exception>` 头文件中声明。不要对这个特性过于看好，要知道，回调函数仍然必须终止程序，否则还是会以某种方式调用 `abort()`。不能只是简单地忽略错误。不过，在退出之前可以使用这个特性打印一条有帮助的错误消息。以下是一个 `main()` 函数的例子，它没有捕获 `readIntegerFile()` 抛出的异常。相反，这个 `main()` 函数只是将 `terminate_handler` 设置为一个回调函数，这个回调函数会在退出之前打印一条错误消息：

```

void myTerminate()
{
    cout << "Uncaught exception!\n";
    exit(1);
}

int main(int argc, char** argv)
{
    vector<int> myInts;
    const string fileName = "IntegerFile.txt";

    set_terminate(myTerminate);

    readIntegerFile(fileName, myInts);

    for (size_t i = 0; i < myInts.size(); i++) {
        cerr << myInts[i] << " ";
    }
    cout << endl;

    return (0);
}

```

尽管这个例子中没有显示，但要知道，`set_terminate()` 设置新的 `terminate_handler` 时会返回原来的

terminate\_handler。terminate\_handler 应用于整个程序，所以需要新的 terminate\_handler 代码一旦结束工作，应该重新设置回原来的 terminate\_handler，这是一种好的风格。在这个例子中，整个程序都需要新的 terminate\_handler，因此重新设置回原来的 terminate\_handler 没有意义。

尽管了解 set\_terminate() 很重要，但是这并非一种很有效的异常处理方法。建议还是应该单独地捕获和处理各个异常，以便提供更准确的错误处理。

### 15.2.5 抛出列表

C++ 允许指定一个函数或方法想要抛出的异常。这个规范（即所指定的异常）称为抛出列表（throw list）或异常规范（exception specification）。以下是前例中的 readIntegerFile() 函数，在此提供了适当的抛出列表：

```
void readIntegerFile(const string& fileName, vector<int>& dest)
    throw (invalid_argument, runtime_error)
{
    // Remainder of the function is the same as before
}
```

抛出列表只是列出了可能由函数抛出的各种异常类型。需要注意，还必须为函数原型提供抛出列表。

```
void readIntegerFile(const string& fileName, vector<int>& dest)
    throw (invalid_argument, runtime_error);
```

不同于 const，异常规范并不是函数或方法签名的一部分。不能只是根据抛出列表中不同的异常来重载一个函数。

如果一个函数或方法没有指定抛出列表，它就能抛出任何异常。在前面 readIntegerFile() 函数的实现中你已经了解这一点了。如果想指定一个函数或方法不抛出任何异常，就需要明确地写一个空的抛出列表，如下所示：

```
void readIntegerFile(const string& fileName, vector<int>& dest)
    throw ();
```

如果觉得这种做法与你的想法相左，实际上有这种观点的并非你一人。不过，最好还是先接受这种做法，继续学习余下的内容吧。

如果函数没有抛出列表，就可以抛出任何类型的异常。如果函数有一个空的抛出列表，则不能抛出任何异常。

#### 意外异常

遗憾的是，C++ 中并不能在编译时保证抛出列表。调用 readIntegerFile() 的代码不必捕获抛出列表中列出的异常。这与其他一些语言（如 Java）中的做法不同，在 Java 中，如果一个函数的抛出列表中指定了可能抛出的异常，就要求调用此函数的其他函数或方法必须捕获这些异常，或者在自己的函数或方法抛出列表中声明这些异常。

另外，可以如下实现 readIntegerFile()。

```
void readIntegerFile(const string& fileName, vector<int>& dest)
    throw (invalid_argument, runtime_error)
{
    throw (5);
}
```

尽管抛出列表指出 `readIntegerFile()` 不会抛出一个 `int`，但这段代码显然抛出了一个 `int`，而且上述代码确实能够编译和运行。不过，结果可能与预想的不同。假设你编写了以下 `main()` 函数，认为可以捕获 `int`。

```
int main(int argc, char** argv)
{
    vector<int> myInts;
    const string fileName = "IntegerFile.txt";

    try {
        readIntegerFile(fileName, myInts);
    } catch (int x) {
        cerr << "Caught int\n";
    }
}
```

运行这个程序而且 `readIntegerFile()` 抛出 `int` 异常时，程序会终止。它不允许 `main()` 捕获 `int`。不过，可以改变这种行为。

抛出列表不能阻止函数抛出未列出的异常类型，但是能阻止异常真正离开函数（到达异常处理程序）。

一个函数抛出了未列在抛出列表中的异常时，C++ 会调用一个特殊的函数 `unexpected()`。`unexpected()` 的内置实现只是调用 `terminate()`。不过，就像能设置自己的 `terminate_handler` 一样，也可以设置自己的 `unexpected_handler`。不同于 `terminate_handler`，在 `unexpected_handler` 中，除了终止程序外，确实还能做些别的事情。你实现的这个版本要么必须抛出一个新异常，要么必须终止程序，它不能只是正常地退出函数。如果抛出了一个新异常，这个异常会取代原来的意外异常，就好像原来抛出的就是这个新异常一样。如果替代后的异常还是未列在抛出列表中，程序就会做以下的某个工作。如果函数的抛出列表指定了 `bad_exception`，则会抛出 `bad_exception`。否则，程序会终止。`unexpected()` 的定制实现通常用于把意外异常转换为预料中的异常。例如，可以如下编写一个 `unexpected()`：

```
void myUnexpected()
{
    cout << "Unexpected exception!\n";
    throw runtime_error("");
}
```

这个代码会把一个意外异常转换为一个 `runtime_error` 异常（函数 `readIntegerFile()` 的抛出列表中有 `runtime_error` 异常）。

可以在 `main()` 中利用 `set_unexpected` 函数设置这个意外异常处理程序。类似于 `set_terminate()`，`set_unexpected()` 会返回当前的处理程序。`unexpected()` 函数应用于整个程序，而不只是这个函数，因此需要特殊处理程序的代码完成工作之后，重新设置回原来的处理程序：

```
int main(int argc, char** argv)
{
    vector<int> myInts;
    const string fileName = "IntegerFile.txt";

    unexpected_handler old_handler = set_unexpected(myUnexpected);
    try {
        readIntegerFile(fileName, myInts);
    }
```

```

    } catch (const invalid_argument& e) {
        cerr << "Unable to open file " << fileName << endl;
        exit (1);
    } catch (const runtime_error& e) {
        cerr << "Error reading file " << fileName << endl;
        exit (1);
    } catch (int x) {
        cout << "Caught int\n";
    }
    set_unexpected(old_handler);
    // Remainder of function omitted
}

```

现在 main() 把 readIntegerFile() 抛出的所有异常都转换为一个 runtime\_error 异常, 这样就能处理 readIntegerFile() 抛出的所有异常。不过, 与 set\_terminate() 类似, 建议你明智地使用这个功能。

unexpected()、set\_unexpected() 和 bad\_exception 都在 <exception> 头文件中声明。

### 在覆盖方法中修改抛出列表

在子类中覆盖一个虚 (virtual) 方法时, 可以修改抛出列表, 只要所修改的抛出列表比超类中的抛出列表更为限定 (more restrictive) 就行。以下修改都认为是更限定的修改:

- 从列表中删除异常
  - 增加超类抛出列表中所出现异常的子类
- 但认为以下修改并非更限定:
- 增加的异常并非超类抛出列表中异常的子类
  - 完全删除抛出列表

在覆盖方法时, 如果修改了抛出列表, 要记住调用超类中方法的代码都必须能够调用子类中的相应方法。因此, 不能增加异常。

例如, 假设有以下超类:

```

class Base
{
public:
    virtual void func() throw(exception) { cout << "Base!\n"; }
};

```

可以编写一个子类覆盖 func(), 并指定它不抛出任何异常:

```

class Derived : public Base
{
public:
    virtual void func() throw() { cout << "Derived!\n"; }
};

```

还可以覆盖 func(), 指出它不仅抛出一个 exception, 还可能抛出一个 runtime\_error, 因为 runtime\_error 是 exception 的一个子类。

```

class Derived : public Base
{
public:

```

```
virtual void func() throw(exception, runtime_error)
{ cout << "Derived!\n"; }
};
```

不过，不能将抛出列表完全删除，因为这表示 func() 可以抛出任何异常。

假设 Base 如下所示：

```
class Base
{
public:
    virtual void func() throw(runtime_error) { cout << "Base!\n"; }
};
```

不能在 Derived 中如下覆盖 func()，使之有下面的抛出列表：

```
class Derived : public Base
{
public:
    virtual void func() throw(exception) { cout << "Derived!\n"; } // ERROR!
};
```

exception 是 runtime\_error 的超类，因此不能用 exception 来代替 runtime\_error。

### 抛出列表有用吗

如果有机会在函数签名中指定它的行为，倘若不利用的话好像有些浪费。从一个特定函数抛出的异常是该函数接口的一个重要部分，对此应该尽可能地提供文档。

遗憾的是，当前使用的大多数 C++ 代码（包括标准库）并没有遵循这个建议。这就使你在这些代码时很难确定可能抛出哪些异常。另外，无法对模板化的函数和方法指定异常特征。如果连用来实例化模板的类型都不知道，就根本没有办法确定这些类型的方法会抛出哪些异常。最后一个问题，抛出列表的语法有些含糊，而且对于抛出列表语法的保证也不明确。

所以，抛出列表到底有没有用，请你自己决定。

## 15.3 异常和多态

如前所述，确实可以抛出任何类型的异常。不过，类是最有用的异常类型。实际上，异常类通常按层次体系编写，这样在捕获异常时就可以利用多态了。

### 15.3.1 标准异常层次体系

你已经见过 C++ 异常层次体系中的几个异常了，如 exception、runtime\_error 和 invalid\_argument。图 15-3 显示了这个完整的层次体系。

C++ 标准库抛出的所有异常都是这个层次体系中类的对象。层次体系中每个类都支持一个 what() 方法，这个方法返回一个描述异常的 char\* 字符串。可以在错误消息中使用这个字符串。

除了基类 exception 外，所有异常类都要求在构造函数中设置 what() 所返回的字符串。这就是为什么必须在构造函数中为 runtime\_error 和 invalid\_argument 指定一个字符串的原因。既然已经知道了这些字符串的作用是什么，就可以让它们有更大作为。下面是一个例子，在此使用这个字符串向调用者传回完整的错误消息：



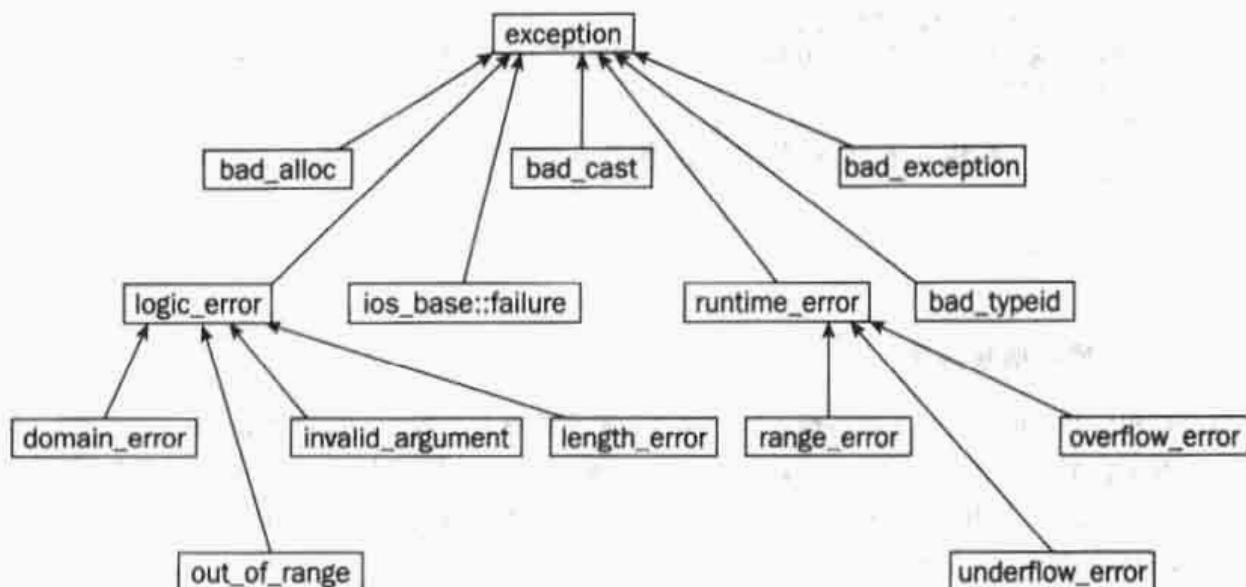


图 15-3

```

void readIntegerFile(const string& fileName, vector<int>& dest)
    throw (invalid_argument, runtime_error)
{

```

```

    ifstream istr;
    int temp;

```

```

    istr.open(fileName.c_str());

```

```

    if (istr.fail()) {

```

```

        // We failed to open the file: throw an exception.

```

```

        string error = "Unable to open file " + fileName;

```

```

        throw invalid_argument(error);
    }

```

```

    // Read the integers one by one and add them to the vector.

```

```

    while (istr >> temp) {
        dest.push_back(temp);
    }

```

```

    if (istr.eof()) {

```

```

        // We reached the end-of-file.

```

```

        istr.close();
    } else {

```

```

        // Some other error. Throw an exception.

```

```

        istr.close();

```

```

        string error = "Unable to read file " + fileName;

```

```

        throw runtime_error(error);
    }
}

```

```

int main(int argc, char** argv)
{

```

```

    // Code omitted

```

```

    try {

```

```

        readIntegerFile(fileName, myInts);
    }
}

```

```

    } catch (const invalid_argument& e) {
        cerr << e.what() << endl;
        exit (1);
    } catch (const runtime_error& e) {
        cerr << e.what() << endl;
        exit (1);
    }
    // Code omitted
}

```

### 15.3.2 按类层次捕获异常

异常层次体系中最令人兴奋的特性就是可以多态地捕获异常。例如，如果在 `main()` 中发现 `readIntegerFile()` 调用后面有两个 `catch` 语句，可以看到除了所处理的异常类不同以外，它们几乎是相同的。方便的是，`invalid_argument` 和 `runtime_error` 都是 `exception` 的子类，因此可以把这两个 `catch` 语句代之以一条对应类 `exception` 的 `catch` 语句：

```

int main(int argc, char** argv)
{
    // Code omitted
    try {
        readIntegerFile(fileName, myInts);
    } catch (const exception& e) {
        cerr << e.what() << endl;
        exit (1);
    }
    // Code omitted
}

```

对应 `exception` 引用的 `catch` 语句可以与 `exception` 的所有子类匹配，这也包括 `invalid_argument` 和 `runtime_error`。注意，在异常层次体系中捕获异常的层次越高，所做的错误处理特定性就越弱。一般应该尽可能在更为特定的层次上捕获异常。

在多态地捕获异常时，要确保按引用捕获异常。如果按值捕获异常，可能会遇到切割问题 (slicing)，在这种情况下会丢失对象的信息。有关“切割问题”的详细内容请见第 10 章。

多态匹配规则以“先来先服务”原则为基础。C++ 会按顺序将异常与各个 `catch` 语句匹配。如果捕获到的异常是某个 `catch` 语句相应类的一个对象，或者是该类子类的一个对象，该异常就会与这条 `catch` 语句匹配，即使后面的 `catch` 语句还有更精确的匹配，也不会影响，仍会取第一个匹配。例如，假设你想明确地捕获 `readIntegerFile()` 的 `invalid_argument`，但是还要保留通用的 `exception` 从而与任何其他异常匹配。正确的做法如下所示：

```

try {
    readIntegerFile(fileName, myInts);
} catch (const invalid_argument& e) { // List the exception subclass first.
    // Take some special action for invalid filenames.
} catch (const exception& e) { // Now list exception
    cerr << e.what() << endl;
    exit (1);
}

```

第一条 `catch` 语句将捕获 `invalid_argument` 异常，第二条则捕获所有其他的异常。不过，如果把这两

条 catch 语句的顺序倒过来,就得不到同样的结果了:

```
try {
    readIntegerFile(fileName, myInts);
} catch (const exception& e) { // BUG: catching superclass first!
    cerr << e.what() << endl;
    exit (1);
} catch (const invalid_argument& e) {
    // Take some special action for invalid filenames.
}
```

如果按这个顺序,只要是 exception 子类的任何异常都会被第一条 catch 语句捕获,永远也到不了第二条 catch 语句。在这种情况下,有些编译器会发出一个警告,不过这可不一定。

### 15.3.3 编写自己的异常类

编写自己的异常类,这有两个优点。

1. C++ 标准库中的异常是有限的。这样可以不使用一个有通用名的异常类(如 runtime\_exception),而是用更有意义的名字创建异常类,即异常类的名字更能反映程序中的某些特定错误。

2. 可以向这些异常增加自己的信息。标准层次体系中的异常只允许设置一个错误字符串。你可能还想在异常中传递不同的信息。

建议你编写的所有异常类都直接或间接地继承自标准 exception 类。如果项目中的每一个人都遵循这个原则,你就能知道程序中的每个异常都会是 exception 的一个子类(假设没有使用违反这个规则的第三方库)。这个原则使得通过多态处理异常容易得多。

例如,invalid\_argument 和 runtime\_error 没有很好地反映 readIntegerFile() 中的文件打开和读取错误。你可以为文件错误定义自己的错误体系,先从一个通用的 FileError 类开始:

```
class FileError : public runtime_error
{
public:
    FileError(const string& fileIn) : runtime_error(""), mFile(fileIn) {}
    virtual ~FileError() throw() {}

    virtual const char* what() const throw() { return mMsg.c_str(); }
    string getFileName() { return mFile; }

protected:
    string mFile, mMsg;
};
```

作为一个好的程序员,应当让 FileError 成为标准异常体系中的一部分。把它作为 runtime\_error 的一个子类集成到标准异常层次体系中来,这看来很合适。在编写 runtime\_error (或者标准层次体系中的任何其他异常)的一个子类时,需要覆盖两个方法: what() 和析构函数。

前面已经给出了 what() 的签名,这个方法要返回一个 char\* 字符串,在对象撤销前这个串都是有效的。对于 FileError,这个字符串取自 mMsg 数据成员,构造函数中此数据成员设置为 ""。FileError 的子类如果想要其他的信息,就必须将这个 mMsg 字符串设置为其他的内容。

还必须覆盖析构函数,从而指定空的抛出列表。编译器生成的析构函数没有抛出列表,这样就不能编译,因为 runtime\_error 指定了一个空的抛出列表。

通用 FileError 类还包含一个文件名,以及对应该文件名的一个存取方法。

如果文件无法打开，这是 `readIntegerFile()` 中出现的第一种异常情况。因此，你可能想编写 `FileError` 的一个 `FileOpenError` 子类：

```
class FileOpenError : public FileError
{
public:
    FileOpenError(const string& fileNameIn);
    virtual ~FileOpenError() throw() {}
};

FileOpenError::FileOpenError(const string& fileNameIn) : FileError(fileNameIn)
{
    mMsg = "Unable to open " + fileNameIn;
}
```

`FileOpenError` 修改了 `mMsg` 字符串来表示文件打开错误。

如果文件无法正确地读取，这是 `readIntegerFile()` 中出现的第二种异常情况。这个异常如果能包含文件中错误所在的行号、文件名以及 `what()` 返回的错误消息字符串，这可能很有用。以下是 `FileError` 的一个 `FileReadError` 子类：

```
class FileReadError : public FileError
{
public:
    FileReadError(const string& fileNameIn, int lineNumIn);
    virtual ~FileReadError() throw() {}
    int getLineNum() { return mLineNum; }

protected:
    int mLineNum;
};

FileReadError::FileReadError(const string& fileNameIn, int lineNumIn) :
    FileError(fileNameIn), mLineNum(lineNumIn)
{
    ostringstream ostr;

    ostr << "Error reading " << fileNameIn << " at line " << lineNumIn;
    mMsg = ostr.str();
}
```

当然，为了正确地设置行号，需要修改 `readIntegerFile()` 函数来记录所读的行数，而不只是直接读取整数。以下是使用了这些新异常的新的 `readIntegerFile()` 函数：

```
void readIntegerFile(const string& fileName, vector<int>& dest)
    throw (FileOpenError, FileReadError)
{
    ifstream istr;
    int temp;
    char line[1024]; // Assume that no line is longer than 1024 characters.
    int lineNumber = 0;

    istr.open(fileName.c_str());
    if (istr.fail()) {
```

```
// We failed to open the file: throw an exception.
throw FileOpenError(fileName);
}

while (!istr.eof()) {
    // Read one line from the file.
    istr.getline(line, 1024);
    lineNumber++;

    // Create a string stream out of the line.
    istringstream lineStream(line);

    // Read the integers one by one and add them to the vector.
    while (lineStream >> temp) {
        dest.push_back(temp);
    }

    if (!lineStream.eof()) {
        // Some other error. Close the file and throw an exception.
        istr.close();
        throw FileReadError(fileName, lineNumber);
    }
}
istr.close();
}
```

现在调用 `readIntegerFile()` 的代码可以使用多态来捕获类型为 `FileError` 的异常，如下所示：

```
try {
    readIntegerFile(fileName, myInts);
} catch (const FileError& e) {
    cerr << e.what() << endl;
    exit (1);
}
```

如果类对象要用作异常，编写这种类有一个技巧。一段代码抛出一个异常时，会复制所抛出的对象或值。也就是说，会使用复制构造函数从原对象构造一个新对象。必须复制这个对象，因为原对象可能在捕获到异常之前出作用域（销毁，并回收其内存），原对象的作用域位于栈中的更高层次。因此，如果编写了一个类，这个类的对象会作为异常抛出，就必须使这些对象是可复制的。这说明，如果动态分配了内存，就必须编写一个析构函数、一个复制构造函数和一个赋值操作符，如第 9 章所述。

作为异常抛出的对象至少会按值复制一次。

异常可能会复制不只一次，但是只有当按值而不是按引用捕获异常时才会出现这种情况。

按引用捕获异常对象来避免不必要的复制。

## 15.4 栈展开和清除

当一段代码抛出一个异常时，控制会立即跳至与该异常匹配的异常处理程序。这个异常处理程序可能位于栈中一个或几个函数调用之上。当控制在栈中上跳时，这个过程称为栈展开（stack unwinding），当前执行点之后各函数中余下的所有代码都会跳过。不过，每个“展开”函数中的局部对象和变量会撤销，就好像代码正常地完成了这个函数一样。

不过，在栈展开过程中，指针变量不会释放，而且不会完成其他清除工作。这就会带来问题，如下代码所示：

```
#include <fstream>
#include <iostream>
#include <stdexcept>
using namespace std;

void funcOne() throw(exception);
void funcTwo() throw(exception);

int main(int argc, char** argv)
{
    try {
        funcOne();
    } catch (exception& e) {
        cerr << "Exception caught!\n";
        exit(1);
    }

    return (0);
}

void funcOne() throw(exception)
{
    string str1;
    string* str2 = new string();
    funcTwo();
    delete str2;
}

void funcTwo() throw(exception)
{
    ifstream istr;
    istr.open("filename");
    throw exception();
    istr.close();
}
```

funcTwo() 抛出一个异常时，最近的异常处理程序位于 main() 中。控制就立即从 funcTwo() 中的这行代码：

```
throw exception();
```

跳至 main() 中的代码行：

```
cerr << "Exception caught! \n";
```

在 funcTwo() 中，控制仍停留在抛出异常的那行代码上，因此后面的这行代码永远也得不到机会运行。

```
istr.close();
```

不过，幸运的是，这里会调用 ifstream 析构函数，因为 istr 是栈上的局部变量。而 ifstream 析构函数会关闭文件，因此这里没有资源泄漏。如果 istr 是动态分配的，它就不会撤销，而文件也不会关闭。

在 funcOne() 中，控制位于 funcTwo() 调用上，因此后面的这行代码永远也得不到机会运行：



```
delete str2;
```

在这种情况下，确实会存在内存泄漏。栈展开不会自动地对 `str2` 调用 `delete`。不过，`str1` 会适当地撤销，因为它是栈上的局部变量。栈展开会正确地撤销所有局部变量。

如果异常处理不够细心，就会导致内存和资源泄漏。

以下是两个处理这个问题的技术。

#### 15.4.1 捕获、清除和重新抛出

要避免内存和资源泄漏，首要的而且也是最常见的技术就是让每个函数都捕获所有可能的异常，完成必要的清除工作，再把异常重新抛给栈上更高的函数来处理。以下是修改后的 `funcOne()`，在此就采用了这个技术：

```
void funcOne() throw(exception)
{
    string str1;
    string* str2 = new string();
    try {
        funcTwo();
    } catch (...) {
        delete str2;
        throw; // Rethrow the exception.
    }
    delete str2;
}
```

这个函数将 `funcTwo()` 调用与异常处理程序“包”在一起，这个异常处理程序将完成清除工作（对 `str2` 调用 `delete`），然后重新抛出这个异常。关键字 `throw` 本身只是重新抛出最近捕获到的异常（无论是什么异常）。注意，`catch` 语句使用了 `...` 语句来捕获任何异常。

这个方法可以很好地工作，不过可能会很混乱。具体地，需要注意的是，这里有两行代码对 `str2` 调用 `delete`，一个用来处理异常，另一个则是函数正确退出时的正常处理。

#### 15.4.2 使用智能指针

利用智能指针，可以编写能够自动防止内存泄漏的代码，即避免通过异常处理而导致内存泄漏。从第 13 章可知，智能指针对象要在栈上分配，因此无论智能指针对象何时撤销，都会在底层哑指针上调用 `delete`。以下是 `funcOne()` 的一个例子，在此使用了标准库中的 `auto_ptr` 智能指针模板类：

```
#include <memory>
using namespace std;

void funcOne() throw(exception)
{
    string str1;
    auto_ptr<string> str2(new string("hello"));
    funcTwo();
}
```

利用智能指针，你不必记住自己来撤销底层的哑指针（对底层哑指针调用 `delete`）：不论是由于异常而退出函数，还是正常地退出函数，智能指针都会完成底层指针的撤销。

## 15.5 常见的错误处理问题

在程序中是否使用异常取决于你和你的同事。不过，我们强烈建议你为程序制定一个正式的错误处理计划（可以使用异常，也可以不使用异常）。如果使用了异常，通常更容易提供一个统一的错误处理机制，不过不用异常也不是全无可能。一个计划好不好，最重要的方面是程序所有模块中错误处理的统一性。要确保项目中每个程序员都理解并遵循错误处理规则。

这一节将讨论异常方面最常见的一些错误处理问题，不过，这些问题对于不使用异常的程序也有关系。

### 15.5.1 内存分配错误

尽管本书到目前为止的例子都忽略了这种可能性，但是内存分配确实可能会失败。不过，成品（生产）代码必须考虑到内存分配失败。C++ 提供了许多不同方法来处理内存错误。

`new` 和 `new []` 如果不能分配内存，其默认行为是抛出一个类型为 `bad_alloc` 的异常，这个类型在 `<new>` 头文件中定义。你的代码应当捕获这些异常，并适当地处理。“适当”的具体定义取决于特定应用。在某些情况下，程序要正确地运行，内存可能至关重要，在这种情况下，打印一条错误消息然后退出是最好的做法。另外一些情况下，内存可能只对某个特定操作或任务是必要的，此时可以打印一条错误消息，并置该特定操作失败，不过程序仍继续运行。

因此，`new` 语句应该如下使用：

```
try {
    ptr = new int[numInts];
} catch (bad_alloc& e) {
    cerr << "Unable to allocate memory!\n";
    // Handle memory allocation failure.
    return;
}
// Proceed with function that assumes memory has been allocated.
```

当然，如果可行，也可以用一个 `try/catch` 块在程序中的一个更高位置成批地处理多个可能的新失败。

还要考虑到，记录一个错误也要分配内存。如果 `new` 失败，甚至可能没有留下足够的内存来记录错误消息。

#### 非抛出 (nothrow) `new`

第 13 章曾提到，如果不喜欢异常，可以回过头去采用原来的 C 模型，其中内存分配例程若无法分配内存，就会返回 `NULL` 指针。C++ 提供了 `new` 和 `new []` 的 `nothrow` 版本，即如果无法分配内存，它们会返回 `NULL` 而不是抛出一个异常：

```
ptr = new(nothrow) int[numInts];
if (ptr == NULL) {
    cerr << "Unable to allocate memory!\n";
    // Handle memory allocation failure.
    return;
}
// Proceed with function that assumes memory has been allocated.
```

这个语法有点奇怪，实际上是把“nothrow”写作为 `new` 的一个参数了（事实上也确实如此）。

### 定制内存分配失败行为

C++允许指定一个 *new* 处理程序 (newhandler) 回调函数。默认地, 并没有 *new* 处理程序, 因此, *new* 和 *new []* 只是抛出 *bad\_alloc* 异常。不过, 如果存在一个 *new* 处理程序, 内存分配例程遇到内存分配失败时就会调用这个 *new* 处理程序, 而不是抛出一个异常。如果 *new* 处理程序返回, 内存分配例程会尝试再次分配内存, 如果失败了还会调用 *new* 处理程序。这个循环会成为一个无限循环, 除非 *new* 处理程序通过以下某种做法使情况有所改变。实际说来, 在这 4 种选择中有些选择更好一些。以下分别列出, 并做相关说明:

- 提供更多可用内存。提供内存空间的一个技巧是在程序开始时分配一大块内存, 然后在 *new* 处理程序中用 *delete* 来释放。如果当前的内存请求需要的内存不超过你在 *new* 处理程序中释放的内存, 内存分配例程现在就可以成功分配内存了。不过, 这种技术的意义并不大。如果先前没有预分配那块内存, 内存请求也可能直接成功, 而没有必要调用 *new* 处理程序。这样做的好处尽管不多, 不过有一点需要指出, 这样一来就可以记录一条 *new* 处理程序中最低内存是多少的警告消息。
- 抛出一个异常。*new* 和 *new []* 有抛出列表, 指出它们只会抛出类型为 *bad\_alloc* 的异常。因此, 除非想创建一个 *unexpected()* 调用, 如果想从 *new* 处理程序抛出一个异常, 就必须抛出 *bad\_alloc* 或子类。不过, 并不需要一个新的 *new* 处理程序来抛出一个异常, 默认行为已经足够了。因此, 如果 *new* 处理程序只是要做这些, 那么完全没有必要编写一个 *new* 处理程序。
- 设置一个不同的 *new* 处理程序。理论上讲, 可以有一系列的 *new* 处理程序, 每个 *new* 处理程序都会创建内存, 如果失败则设置一个不同的 *new* 处理程序。不过, 这种情况下, 往往复杂性大于实用性。
- 终止程序。这种选择是四者当中最实际、最有用的一种。*new* 处理程序可以简单地记录一个错误消息, 并终止程序。相对于捕获一个 *bad\_alloc*, 并在异常处理程序中终止程序的做法, 使用一个 *new* 处理程序的好处在于, 可以把失败处理都集中到一个函数中, 这样就不需要用 *try/catch* 块把代码弄得支离破碎。如果存在一些内存分配失败, 不过仍允许程序继续运行, 在这种情况下, 调用 *new* 之前可以简单地把 *new* 处理程序临时设置回默认的 *NULL*。

如果未在 *new* 处理程序中采取以上某一种动作, 所有内存分配失败都会导致无限循环。

可以用一个 *set\_new\_handler()* 调用来设置 *new* 处理程序, 这个函数在 *<new>* 头文件中声明。*set\_new\_handler()* 只是 C++ 设置回调函数的三个函数中的一个。另外两个分别是 *set\_terminate()* 和 *set\_unexpected()*, 见本章前面的讨论。以下是一个 *new* 处理程序的例子, 在此记录了一条错误消息, 并中止程序:

```
void myNewHandler()
{
    cerr << "Unable to allocate memory. Terminating program!\n";
    abort();
}
```

*new* 处理程序没有任何参数, 而且不返回任何值。这个 *new* 处理程序调用 *<cstdlib>* 中的 *abort()* 函数来终止程序。

可以如下设置 *new* 处理程序:

```
#include <new>
#include <cstdlib>
#include <iostream>
```

```
using namespace std;

int main(int argc, char** argv)
{
    // Code omitted

    // Set the new new_handler and save the old.
    new_handler oldHandler = set_new_handler(myNewHandler);
    // Code that calls new

    // Reset the old new_handler.
    set_new_handler(oldHandler);
    // Code omitted
    return (0);
}
```

注意，new\_handler 是函数指针类型的一个类型定义 (typedef)，set\_new\_handler() 就取此类型作为参数。

### 15.5.2 构造函数中的错误

C++ 程序员了解异常之前，通常被错误处理和构造函数搞得一头雾水。如果一个构造函数无法适当地构造对象会怎么样？构造函数没有返回值，因此标准的非异常错误处理机制无法工作。如果没有异常，最好的做法就是在对象中设置一个标志，指示它未能正确地构造。可以提供名为 checkConstructionStatus() 之类的方法，此方法返回该标志的值，而且希望客户构造了对象之后能够记得对对象调用这个函数。

异常则提供了一个好得多的解决方案。可以从一个构造函数抛出一个异常，即使无法返回一个值也没有关系。利用异常，能很容易地告诉客户对象的构造是否成功。不过，这里有一个重要问题：如果由异常退出构造函数，该对象的析构函数永远也得不到调用。因此，你必须很小心，在允许异常退出构造函数之前，必须清除所有资源，并释放构造函数中分配的所有内存。这个问题对于其他函数也是一样的，只不过在构造函数中这个问题更微妙一些，因为你可能习惯了让析构函数来负责内存的撤销和资源的释放。

以下是第 11 章中 GameBoard 类的构造函数例子，在此有所修改以利用异常处理：

```
GameBoard::GameBoard(int inWidth, int inHeight) throw(bad_alloc) :
    mWidth(inWidth), mHeight(inHeight)
{
    int i, j;
    mCells = new GamePiece* [mWidth];

    try {
        for (i = 0; i < mWidth; i++) {
            mCells[i] = new GamePiece[mHeight];
        }
    } catch (...) {
        //
        // Clean up any memory we already allocated, because the destructor
        // will never be called. The upper bound of the for loop is the index
        // of the last element in the mCells array that we tried to allocate
        // (the one that failed). All indices before that one store pointers to
        // allocated memory that must be freed.
    }
}
```

```

//
for (j = 0; j < i; j++) {
    delete [] mCells[j];
}
delete [] mCells;

// Translate any exception to bad_alloc.
throw bad_alloc();
}
}

```

如果第一个 new 抛出一个异常，没有关系，因为构造函数尚未分配任何需要释放的东西。不过，如果后面的某个 new 调用抛出了异常，构造函数就必须清除已经分配的所有内存。在此通过……来捕获任何异常，因为它不知道 GamePiece 构造函数本身可能抛出哪些异常。

你可能想知道增加了继承后会出现什么情况。超类构造函数会在子类构造函数之前先运行。如果子类构造函数抛出一个异常，那么超类构造函数分配的资源如何释放呢？答案是 C++ 将确保所有完全构造的“子对象”的析构函数都会运行。因此，只要是正常完成的构造函数（没有异常）都会导致相应的析构函数得到运行。

### 15.5.3 析构函数中的错误

应当在析构函数本身处理析构函数中出现的所有错误条件。不要从析构函数抛出任何异常，对此有三个原因：

1. 在栈展开的过程中，即使有一个未解决的异常，析构函数也能运行。如果你从析构函数抛出一个异常，而此时还存在另一个异常，程序会终止。很新颖地，C++ 提供了一种能力，可以确定是因为一个正常的退出或撤销函数调用来执行析构函数，还是由于栈展开而执行析构函数。<exception> 头文件中声明有一个函数 `uncaught_exception()`，如果有一个未捕获的异常，而且你正处在栈展开的过程中，它就会返回 true，否则，此函数返回 false。不过，这种方法很混乱，应当着力避免。

2. 客户会采取什么动作？客户不会显式地调用析构函数，它们会调用 delete，而 delete 会调用析构函数。如果从析构函数抛出一个异常，客户会怎么做呢？它不能再对对象调用 delete 了，而且不能显式地调用析构函数。客户采用哪一种动作都不合理，因此根本不当为此代码加上异常处理的负担。

3. 析构函数是释放对象中所用内存和资源的一个机会。如果浪费了这个机会，由于出现异常而过早地退出函数，就无法回收和释放内存或资源了。

因此，要非常小心，对于析构函数中所做调用可能抛出的任何异常，都必须在析构函数中捕获。正常情况下，析构函数只能调用 delete 和 delete []，而它们不能抛出任何异常，因此这不算大问题。

## 15.6 综合

既然已经了解了错误处理和异常，以下修改了第 11 章中完整的 GameBoard 类，在此使用了异常。首先，下面是类定义：

```

#include <stdexcept>
#include <new>
using std::bad_alloc;
using std::out_of_range;

class GameBoard
{

```



```

public:
    GameBoard(int inWidth = kDefaultWidth, int inHeight = kDefaultHeight)
        throw(bad_alloc);
    GameBoard(const GameBoard& src) throw(bad_alloc);
    ~GameBoard() throw();
    GameBoard& operator=(const GameBoard& rhs) throw(bad_alloc);

    void setPieceAt(int x, int y, const GamePiece& inPiece)
        throw(out_of_range);
    GamePiece& getPieceAt(int x, int y) throw(out_of_range);
    const GamePiece& getPieceAt(int x, int y) const throw(out_of_range);

    int getHeight() const throw() { return mHeight; }
    int getWidth() const throw() { return mWidth; }

    static const int kDefaultWidth = 100;
    static const int kDefaultHeight = 100;

protected:
    void copyFrom(const GameBoard& src) throw(bad_alloc);

    GamePiece** mCells;
    int mWidth, mHeight;
};

```

构造函数和 `operator=` 都抛出 `bad_alloc`，因为它们会完成内存分配。析构函数 `getHeight()` 和 `getWidth()` 不抛出任何异常。`setPieceAt()` 和 `getPieceAt()` 在调用者提供的宽度和高度不合法的情况下会抛出 `out_of_range`。

在上一节中已经看到了构造函数的实现。以下是带异常处理的 `copyFrom()`、`setPieceAt()` 和 `getPieceAt()` 方法的实现。复制构造函数和 `operator=` 的实现除了抛出列表外都不变，因为所有工作都在 `copyFrom()` 中完成，因此这里不再给出其实现。析构函数也不变，因此其实现也不再提供。有关细节请参考第 11 章。

```

void GameBoard::copyFrom(const GameBoard& src) throw(bad_alloc)

```

```

{
    int i, j;
    mWidth = src.mWidth;
    mHeight = src.mHeight;

    mCells = new GamePiece *[mWidth];

    try {
        for (i = 0; i < mWidth; i++) {
            mCells[i] = new GamePiece[mHeight];
        }
    } catch (...) {
        // Clean up any memory we already allocated.
        // If this function is called from the copy constructor,
        // the destructor will never be called.
        // Use the same upper bound on the loop as described in the constructor.
        for (j = 0; j < i; j++) {
            delete [] mCells[j];
        }
        delete [] mCells;
    }
}

```



```

        // Set mCells and mWidth to values that will allow the
        // destructor to run without harming anything.
        // This function is called from operator=, in which case the
        // object was already constructed, so the destructor will be
        // called.
        mCells = NULL;
        mWidth = 0;
        throw bad_alloc();
    }

```

```

    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}

```

```

void GameBoard::setPieceAt(int x, int y, const GamePiece& inElem)
    throw(out_of_range)
{

```

```

    // Check for out of range arguments.
    if (x < 0 || x >= mWidth || y < 0 || y >= mHeight) {
        throw out_of_range("Invalid width or height");
    }

```

```

    mCells[x][y] = inElem;
}

```

```

GamePiece& GameBoard::getPieceAt(int x, int y) throw(out_of_range)
{

```

```

    // Check for out of range arguments.
    if (x < 0 || x >= mWidth || y < 0 || y >= mHeight) {
        throw out_of_range("Invalid width or height");
    }

```

```

    return (mCells[x][y]);
}

```

```

const GamePiece& GameBoard::getPieceAt(int x, int y) const throw(out_of_range)
{

```

```

    // Check for out of range arguments.
    if (x < 0 || x >= mWidth || y < 0 || y >= mHeight) {
        throw out_of_range("Invalid width or height");
    }

```

```

    return (mCells[x][y]);
}

```

## 15.7 小结

本章讨论的问题与 C++ 程序中的错误处理有关，并强调了在设计 and 编写程序时必须有一个错误处理计划。通过阅读本章，你了解了 C++ 异常语法和行为的详细内容。本章还介绍了错误处理举足轻重的一些领域，包括 I/O 流、内存分配、构造函数和析构函数。最后，你还看到了 GameBoard 类中错误处理的例子。

下面几章将继续探讨 C++ 语言中的一些高级主题。第 16 章将讨论重载。第 17 章介绍 C++ 中的性能问题。第 18 章教你如何如何将 C++ 与其他语言结合，并在多个平台上运行程序。



# 第四部分

## 确保无错代码

### 第16章 重载 C++ 操作符

C++ 允许为类重新定义操作符的含义，如 +、- 和 =。许多面向对象语言不提供这个功能，所以你可能对它在 C++ 中有多大用处不以为然。不过，类如果能与 int 和 double 之类的内置类型有相同的表现，可能很有好处。甚至还能编写出像数组、函数或指针的类来。

第 3 章和第 5 章分别介绍过面向对象设计和操作符重载。第 8 章和第 9 章提供了对象和基本操作符重载的语法细节。本章将介绍第 9 章未谈到的操作符重载内容。STL 中就大量使用了操作符重载，STL 在第 4 章就提到过，并将在第 21 章~第 23 章详细介绍。在阅读第 21 章~第 23 章之前，应当先阅读并理解本章的内容。

这一章强调了操作符重载的语法和基本语义。在此为大多数操作符都提供了实用示例，只有少数有所例外，这些操作符的实用示例将在后面的章节中给出。

第 9 章包含的信息本章将不再介绍。

本章内容包括：

- 操作符重载概述
  - 重载操作符的基本原理
  - 操作符重载中的限制、警告和选择
  - 可以重载、不能重载和不应重载的操作符小结
- 如何重载一元加号、一元减号、自增和自减操作符
- 如何重载 I/O 流操作符 (operator<< 和 operator>>)
- 如何重载下标 (数组索引) 操作符
- 如何重载函数调用操作符
- 如何重载解除引用操作符 (\* 和 ->)
- 如何编写转换操作符
- 如何重载内存分配和撤销操作符

#### 16.1 操作符重载概述

第 1 章已经复习过，C++ 中的操作符就是一些符号，如 +、<、\* 和 <<。在内置类型 (如 int 和 double) 上使用这些操作符可以完成算术、逻辑和其他操作。还有一些操作符 (如 -> 和 &) 允许对指

针解除引用。C++中操作符的概念范围很广，甚至还包括 `[]`（数组索引）、`()`（函数调用）和内存分配和撤销例程。

通过操作符重载，可以针对类改变操作符的行为。不过，这种能力也带来了一些规则、限制和选择。

### 16.1.1 为什么要重载操作符

在学习如何重载操作符之前，你可能想知道为什么要重载操作符。对于不同的操作符，原因有所不同，不过一般的指导原则是让类表现得像内置类型一样。类越像内置类型，客户使用起来就越容易。例如，如果想编写一个类来表示小数，倘若能对`+`、`-`、`*`和`/`应用于该类时的含义有所定义，这会有很大帮助。

重载操作符还有一个原因，这就是能够对程序中的表现有更多控制。例如，可以为类重载内存分配和撤销例程，以指定对于每个新对象究竟该如何分配和回收内存。

要强调重要的一点，操作符重载并不一定能让类的开发人员更轻松；其主要目的是使类客户的日子更好过。

### 16.1.2 操作符重载的限制

在重载操作符时以下所列的事情不能做：

- 不能增加新的操作符符号。只能对语言中已有的操作符重新定义其含义。16.1.5节中的表16-1列出了所有可以重载的操作符。
- 有些操作符不能重载，如，`·`（对象中的成员访问）、`::`（作用域解析操作符）、`sizeof`、`?:`（三元操作符），等等。16.1.5节中的表16-1列出了能够重载的所有操作符。不能重载的操作符通常你就根本不想重载，所以这条限制对你来说可能不算是真正的限制。
- 不能改变操作符的元数（arity）。元数是指与操作符有关的参数或操作数（operand）个数。一元操作符（如`+`）只应用于一个操作数。二元操作符（如`+`）只能作用于两个操作数。另外只有一个三元操作符`?:`。在重载`[]`（数组中括号，或数组索引操作符）时，这个限制可能会带来麻烦，后面将介绍有关内容。
- 不能改变操作符的优先级（precedence）或关联性（associativity）。这条规则确定了语句中操作符采用何种顺序计算。同样地，在大多数程序中这个约束都不会带来问题，因为修改计算顺序一般没有什么好处。
- 不能重新定义内置类型的操作符。可以重载的操作符必须是类中的一个方法，或者至少一个全局重载操作符函数的某个参数必须是一个用户定义类型（如一个类）。这说明，不能做一些奇怪的事情，如重新定义`int`的`+`操作符，让它完成减法操作，不过对于自己的类，定义这样一个完成减法的`+`操作符是可以的。这条规则有一个例外，就是内存分配和撤销例程。可以替换程序中所有内存分配的全局例程。

有些操作符本身就有两个不同的含义。例如，`-`操作符不仅可以用作一个二元操作符（如`x = y - z`），也可以用作作为一个一元操作符（如`x = -y`；）。`*`操作符可以用作乘法操作符，也可以用于对一个指针解除引用。`<<`既是插入操作符，也是左移操作符，这要取决于上下文。也可以对操作符重载，使之有两重含义。

### 16.1.3 操作符重载中的选择

在重载一个操作符时，要编写一个名字形如`operatorX`的函数或方法，在此`X`是某操作符的符号。例如，在第9章中，已经看到为`SpreadsheetCell`对象声明的`operator+`，如下所示。

```
friend const SpreadsheetCell operator+ (const SpreadsheetCell& lhs,  
    const SpreadsheetCell& rhs);
```

在你编写的每个重载操作符函数或方法中有许多选择。

### 方法或全局函数

首先，必须确定你的操作符到底要作为你的类的一个方法，还是要作为一个全局函数（通常是这个类的一个友元）。你如何选择呢？首先，需要理解这两个选择之间的差别。如果操作符是一个类的方法，该操作符相应表达式的左边必须是此类的一个对象。如果编写一个全局函数，左边则可以是另一种不同类型的对象。

存在三种不同类型的操作符：

- 必须是方法的操作符。C++ 语言要求某些操作符必须是类的方法，因为在类之外这些操作符没有意义。例如，`operator=` 与类就紧密绑定，因此不会在别处存在。表 16-1 列出了必须是方法的操作符。对于这些操作符，要选择作为一个方法还是作为全局函数就很简单！不过，大多数操作符都没有这个要求。
- 必须是全局函数的操作符。如果需要对操作符左边的操作数是另一种类型的变量（不同于类），就必须让该操作符作为一个全局函数。这条规则特别适用于 `operator<<` 和 `operator>>`，其左边操作数是一个 `ostream` 对象，而不是类的对象。另外，满足交换律的操作符（如二元操作符 `+` 和 `-`）应当允许左边的变量不是类的对象。这个问题在第 9 章解释过。
- 既可以是方法又可以是全局函数的操作符。在 C++ 群体中，对于编写方法还是编写全局函数来重载操作符，其中哪一种做法更好，存在着一些分歧。不过，我们建议以下原则：让每一个操作符都作为一个方法，除非根据前面的要求必须作为全局函数才有所例外。这个原则的一个主要优点在于，方法可以是虚方法（`virtual`），但友元函数则不能是虚的。因此，计划在一个继承树中编写重载操作符时，应当尽可能地将其实现为方法。

将重载操作符编写为方法时，如果它不会改变对象，则应将整个方法标记为 `const`。这样一来，就可以在 `const` 对象上调用此操作符了。

### 选择参数类型

在选择参数类型时往往存在某些限制，因为你不能改变参数的个数（不过也有例外，本章后面将解释这个问题）。例如，`operator+` 作为一个全局函数时总是有两个参数；如果它是一个方法，则有一个参数。如果与此标准不符，编译器就会发出一个错误。从这个意义上说，操作符函数与常规的函数有所区别，因为可以用不同个数的参数重载常规函数。另外，尽管可以为任何类型编写操作符，但操作符的参数类型仅限于当前类（就是要为这个类编写操作符）。如果想为类 `T` 实现加法，就不能编写一个取两个 `string` 的 `operator+`。如果要确定究竟是按值取参数还是按引用取参数，以及是否该置其为 `const`，这就需要选择了。

选择按值还是按引用很容易，每个参数都应当按引用传递。如第 9 章和第 12 章所述，如果可以按引用传递就不要按值传递对象。

要不要 `const`，这个决定也很简单：置每个参数都为 `const`，除非确实要修改此参数。表 16-1 列出了每个操作符的示例原型，并适当地提供了标以 `const` 并作为引用传递的参数。

### 选择返回类型

应该记得，C++ 不会根据返回类型确定重载解析。因此，在编写重载操作符时，可以指定任何返回类型。不过，只是因为你能做，并不代表应当这样做。这种灵活性意味着你可以编写一些让人混淆的代

码，比如其中的比较操作符返回指针，而算术操作符返回 `bool`！不过，事实上不该这么做。相反，应该适当地编写重载操作符，使之与内置类型相应操作符有相同的表现，返回相同的类型。如果编写了一个比较操作符，则返回一个 `bool`。如果编写了一个算术操作符，则返回表示算术运算结果的对象。有时，最初返回类型并不明显。例如，第 8 章曾谈到，`operator=` 应当返回调用此操作符的对象的引用，以便支持嵌套赋值。其他操作符也有类似的复杂返回类型，这些都在表 16-1 中做了总结。

前面有关参数类型的讨论中涉及引用和 `const` 的选择，对于返回类型也存在这些选择。不过，对于返回值，做出选择更为困难。选择按值还是按引用返回时，一般原则是尽可能返回引用，否则才返回值。怎么知道什么时候可以返回一个引用呢？这个选择只应用于返回对象的操作符：对于返回 `bool` 的比较操作符、没有返回类型的转换操作符以及函数调用操作符（可以返回任何类型），这个选择没有实际意义。如果操作符构造了一个新对象，就必须按值返回这个新对象。如果操作符没有构造新对象，就可以返回调用此操作符的对象（或者是某个参数）的一个引用。表 16-1 中显示了一些例子。

可以作为左值（lvalue，赋值表达式的左边）修改的返回值必须是非 `const`。否则，就应当是 `const`。你可能最初没有想到，很多操作符都会返回左值，包括所有赋值操作符（`operator=`、`operator+=`、`operator-=` 等等）。

如果你对适当的返回类型还有疑惑，可以参考表 16-1。

### 选择行为

在一个重载操作符中可以提供你希望的任何实现。例如，可以编写一个 `operator+` 来启动一个乱画（Scrabble）游戏。不过，正如第 5 章所述，一般应该把操作符的实现限制为客户所预想的行为。应当适当地编写 `operator+`，使它完成加法或类似加法的操作，如串连接。

这一章解释了应当如何实现重载操作符。在一些例外情况下，也可能不遵循这些建议而另辟蹊径，不过，一般而言，遵循这些标准模式才是上策。

#### 16.1.4 不应重载的操作符

有一些操作符要是重载可不是好主意，尽管这些操作符确实允许重载。具体地，重载取地址操作符（`operator&`）就没什么特别的用处，而且如果这样做还会导致混淆，因为可能会以意料之外的方式改变 C++ 语言为此操作符提供的基本行为（取变量的地址）。

另外，还应当避免重载二元布尔操作符 `operator&&` 和 `operator||`，因为你不能保证 C++ 的短路计算原则。

最后一点，不对逗号操作符（`operator,`）重载。是的，你没有看错。C++ 中确实有一个逗号操作符。这也称为序列操作符（sequencing operator），用于分隔同一语句中的两个表达式，并保证这两个表达式按从左到右的顺序计算。重载这个操作符很少有（甚至根本没有）合适的理由。

#### 16.1.5 可重载操作符小结

表 16-1 列出了可以重载的所有操作符，并指定了它们要作为类的方法还是作为全局友元函数，在此说明了应当何时（或何时不应当）重载这些操作符，并提供了一些示例原型，并显示了正确的返回值。

如果将来你打算坐下来编写一个重载操作符，这个表将成为一个很有用的参考。你肯定会忘记应该使用哪个返回类型，以及究竟应该实现为一个函数还是一个方法。希望你能像我们一样经常参考这个表。

在这个表中，T 是要为之编写重载操作符的类的类名，E 是另一个类。



表 16-1

操 作 符	名字或种类	方法还是全局友元函数	何时重载	示例原型
operator+ operator- operator* operator/ operator%	二元算术操作符	建议为全局友元函数	想要为类提供这些操作时	friend const T operator+ (const T&, const T&);
operator- operator+ operator~	一元算术操作符 和位操作符	建议为方法	想要为类提供这些操作时	const T operator- () const;
operator++ operator--	自增和自减操作符	建议为方法	重载二元+和一时	T& operator+ + (); const T operator+ + (int);
operator=	赋值操作符	必须是方法	在对象中动态分配了内存而且想避免赋值时, 如第 9 章所述	T& operator= (const T&);
operator+= operator-= operator*= operator/=. operator% =	简写算术赋值操作符	建议为方法	重载了二元算术操作符时	T& operator+ = (const T&);
operator<< operator>> operator& operator  operator^	二元位操作符	建议为全局友元函数	想要提供这些操作时	friend const T operator<< (const T&, const T&);
operator<<= operator>>= operator&= operator = operator^=	简写位赋值操作符	建议为方法	重载了二元位操作符时	T& operator<< = (const T&);
operator< operator> operator<= operator>= operator==	二元比较操作符	建议为全局友元函数	想要提供这些操作时	friend bool operator< (const T&, const T&);
operator<<< operator>>>	I/O 流操作符 (插入和析取)	建议为全局友元函数	想要提供这些操作时	friend ostream &operator<< (ostream&, const T&); friend istream &operator>> (istream&, T&);

(续)

操 作 符	名字或种类	方法还是全局 友元函数	何 时 重 载	示 例 原 型
operator!	布尔取反操作符	建议为成员方法	很少重载, 可以转而使用 bool 或 void * 转换	bool operator! () const;
operator&& operator	二元布尔操作符	建议为全局友元 函数	很少	friend bool operator&& (const T& lhs, const T& rhs); friend bool operator   (const T& lhs, const T& rhs);
operator[]	下标 (数组索引) 操作符	必须是方法	想在数组型的类中提供下 标索引时	E& operator [] (int); const E& operator [] (int) const;
operator()	函数调用操作符	必须是方法	希望对象有函数指针的表 现时	Return type and arguments can vary; see examples in this chapter
operator new operator new[]	内存分配例程	建议为方法	想要控制类的内存分配时 (很少需要)	void* operator new (size_t size) throw (bad_alloc); void* operator new [] (size_t size) throw (bad_alloc);
operator delete operator delete[]	内存撤销例程	建议为方法	重载了内存分配例程时	void operator delete (void* ptr) throw(); void operator delete [] (void* ptr) throw();
operator* operator->	解除引用操作符	对于 operator - > 必须是方法对于 operator* 建议为 方法	对智能指针很有用	E& operator* () const; E* operator-> () const;
operator&	取地址操作符	N/A	从不	N/A
operator->*	解除指针引用, 指向成员	N/A	从不	N/A
operator,	逗号操作符	N/A	从不	N/A
operator type()	转换或强制转换 操作符 (对于每种 类型分别有单独的 操作符)	必须是方法	想提供从类到其他类型的 转换时	operator type() const;

## 16.2 重载算术操作符

在第 9 章中,你了解了如何编写二元算术操作符和简写算术操作符。不过,第 9 章还没有介绍如何重载所有算术操作符。

### 16.2.1 重载一元减和一元加

C++ 有许多一元算术操作符。一元减和一元加就是其中的两个。你可能用过一元减,但是对于一元加可能还不太清楚。以下是对 `int` 使用这些操作符的例子:

```
int i, j = 4;
i = -j; // Unary minus
i = +i; // Unary plus
j = +(-i); // Apply unary plus to the result of applying unary minus to i.
j = -(-i); // Apply unary minus to the result of applying unary minus to i.
```

一元减对操作数取负,而一元加则直接返回操作数。一元加或一元减并非左值,不能对其赋值。这说明,在重载一元减和一元加时必须返回一个 `const` 对象。不过,要注意,可以对一元加或一元减的结果应用一元加或一元减。由于可以对一个 `const` 临时对象应用这些操作,因此必须置 `operator-` 和 `operator+` 本身为 `const`。否则,编译器不允许对 `const` 临时对象调用这些操作。

以下是一个 `SpreadsheetCell` 类定义的例子,在此提供了重载的 `operator-`。一元加通常是一个无动作的操作 (no-op),因此这个类没有重载一元加。

```
class SpreadsheetCell
{
public:
    // Omitted for brevity. Consult Chapter 9 for details.

    friend const SpreadsheetCell operator+(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator-(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator*(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    friend const SpreadsheetCell operator/(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);
    SpreadsheetCell& operator+=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator-=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator*=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator/=(const SpreadsheetCell& rhs);
    const SpreadsheetCell operator-() const;

protected:
    // Omitted for brevity. Consult Chapter 9 for details.
};
```

以下是一元 `operator-` 的定义。

```
const SpreadsheetCell SpreadsheetCell::operator-() const
{
    SpreadsheetCell newCell(*this);
    newCell.set(-mValue); // call set to update mValue and mStr

    return (newCell);
}
```

`operator-` 不会改变操作数，因此这个方法必须构造一个对原值取负的新 `SpreadsheetCell`，并返回该对象的一个副本。因此，它不能返回一个引用。

### 16.2.2 重载自增和自减

应该记得，让一个变量加 1 有 4 种方法：

```
i = i + 1;
i += 1;
++i;
i++;
```

后两种方法称为自增 (increment) 操作符。第一种形式是前缀自增 (prefix increment)，即先让变量加 1，再返回自增后的新值，用于余下的表达式中。后一种形式为后缀自增 (postfix increment)，它会返回原来的 (未自增的) 值，在余下的表达式中使用。自减操作符的工作也类似。

在对 `operator++` 和 `operator--` (前缀和后缀) 重载时，由于存在两种可能的含义，因此会带来一个问题。例如，编写一个重载的 `operator++` 时，如何指定你在重载前缀版本还是后缀版本呢？C++ 引入了一种技巧可以加以区别：`operator++` 和 `operator--` 的前缀版本不带参数，而后缀版本要取一个类型为 `int` 的参数 (但并不用这个参数)。

如果想为 `SpreadsheetCell` 类重载这些操作符，原型可能如下所示：

```
class SpreadsheetCell
{
public:
    // Omitted for brevity. Consult Chapter 9 for details.
    SpreadsheetCell& operator++(); // Prefix
    const SpreadsheetCell operator++(int); // Postfix
    SpreadsheetCell& operator--(); // Prefix
    const SpreadsheetCell operator--(int); // Postfix
protected:
    // Omitted for brevity. Consult Chapter 9 for details.
};
```

C++ 标准指定自增和自减的前缀版本返回一个左值，因此它们不能返回 `const` 值。前缀形式的返回值与操作数的最后值相同，因此前缀自增和自减可以返回调用此操作符的对象的一个引用。不过，自增和自减的后缀版本返回的值与操作数的最后值不同，因此它们不能返回引用。

以下是这些操作符的实现：

```
SpreadsheetCell& SpreadsheetCell::operator++()
{
    set(mValue + 1);
    return (*this);
}

const SpreadsheetCell SpreadsheetCell::operator++(int)
{
    SpreadsheetCell oldCell(*this); // Save the current value before incrementing
    set(mValue + 1); // Increment
    return (oldCell); // Return the old value.
}

SpreadsheetCell& SpreadsheetCell::operator--()
```

```

    set(mValue - 1);
    return (*this);
}

const SpreadsheetCell SpreadsheetCell::operator--(int)
{
    SpreadsheetCell oldCell(*this); // Save the current value before incrementing
    set(mValue - 1); // Increment
    return (oldCell); // Return the old value.
}

```

现在可以按你的想法完成 SpreadsheetCell 对象的自增和自减了！

```

SpreadsheetCell c1, c2;
c1.set(4);
c2.set(4);

c1++;
++c1;

```

应该记得，自增和自减同样可以作用于指针。在编写类时，如果类是智能指针或迭代器，就可以重载 operator++ 和 operator-- 来提供指针自增和自减。有关内容可以参见第 23 章，在第 23 章中你将了解到如何编写自己的 STL 迭代器。

### 16.3 重载位操作符和二元逻辑操作符

位操作符与算术操作符很类似，位简写赋值操作符则与简写算术赋值操作符很类似。不过，它们都不太常用，因此在这里不再提供例子。表 16-1 提供了一些示例原型，如果确实需要重载这些操作符，通过参考那个表就能很容易地实现。

逻辑操作符要麻烦一些。我们不建议重载 && 和 ||。这些操作符并不真正应用于单个类型，它们会“累计”布尔表达式的结果。另外，会丧失短路计算特性。因此，对特定类型重载这些操作符一般没有什么意义。

### 16.4 重载插入和析取操作符

在 C++ 中，不仅会对算术操作使用操作符，还会用操作符来完成从流读取和写至流的操作。例如，在编写 int 和 string 来输出 (cout) 时，会使用插入操作符 <<：

```

int number = 10;
cout << "The number is " << number << endl;

```

在从流读取时会使用析取操作符 >>：

```

int number;
string str;
cin >> number >> str;

```

也可以编写适用于类的插入和析取操作符，这样就能如下读写类对象：

```

SpreadsheetCell myCell, anotherCell, aThirdCell;

cin >> myCell >> anotherCell >> aThirdCell;
cout << myCell << " " << anotherCell << " " << aThirdCell << endl;

```

在编写插入和析取操作符之前，需要确定希望类如何“流出”，以及如何“读入”。对于 SpreadsheetCell，读写 string 是合理的，因为所有 double 都可以读作为 string（还能转换回 double），但是反过来不行。

插入或析取操作符左边的对象是 istream 或 ostream（如 cin 或 cout），而不是一个 SpreadsheetCell 对象。因为你不能向 istream 或 ostream 增加方法，因此应当将插入和析取操作符编写为 SpreadsheetCell 类的全局友元函数。SpreadsheetCell 类中这些函数的声明如下：

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    friend ostream& operator<<(ostream& ostr, const SpreadsheetCell& cell);
    friend istream& operator>>(istream& istr, SpreadsheetCell& cell);
    // Omitted for brevity
};
```

插入操作符取 ostream 的一个引用作为第一个参数，这样就能将其用于文件输出流、串输出流、cout 和 cerr。有关详细内容请见第 14 章。类似地，通过让析取操作符取 istream 的一个引用作为参数，就可以应用于文件输入流、串输入流和 cin。

operator<< 和 operator>> 的第二个参数是希望读写的 SpreadsheetCell 对象的一个引用。插入操作符不会改变它写的 SpreadsheetCell，因此这个引用可以是 const。不过，析取操作符会修改 SpreadsheetCell 对象，这就要求这个参数是一个非 const 的引用。

这两个操作符都会返回作为第一个参数所给定的流的一个引用，从而可以嵌入对此操作符的调用。要记住，这个操作符语法是显式调用全局 operator>> 或 operator<< 函数的简写。考虑下面这行代码：

```
cin >> myCell >> anotherCell >> aThirdCell;
```

它实际上是以下这行代码的简写：

```
operator>>(operator>>(operator>>(cin, myCell), anotherCell), aThirdCell);
```

可以看到，第一个 operator>> 调用的返回值要用作下一个调用的输入。因此，必须返回流引用，以使用在下一个嵌套调用中。否则，嵌套调用就无法编译。

以下是 SpreadsheetCell 类的 operator<< 和 operator>> 的实现：

```
ostream& operator<<(ostream& ostr, const SpreadsheetCell& cell)
{
    ostr << cell.mString;
    return (ostr);
}

istream& operator>>(istream& istr, SpreadsheetCell& cell)
{
    string temp;
    istr >> temp;
    cell.set(temp);
    return (istr);
}
```

这些函数最麻烦的地方在于，为了正确地设置 mValue，operator>> 必须记住对 SpreadsheetCell 调



用 `set()` 方法，而不是直接设置 `mString`。

## 16.5 重载下标操作符

暂且假设你从来没听说过 STL 中的向量模板类，所以你决定编写自己的动态分配数组类。这个类要允许设置和获取指定索引处的元素，而且要在“后台”负责所有的内存分配。对于一个动态分配的整数数组，第一次尝试编写的类定义可能如下所示：

```
class Array
{
public:
    // Creates an array with a default size that will grow as needed.
    Array();
    ~Array();

    // Returns the value at index x. If index x does not exist in the array,
    // throws an exception of type out_of_range.
    int getElementAt(int x) const;

    // Sets the value at index x to val. If index x is out of range,
    // allocates more space to make it in range.
    void setElementAt(int x, int val);

protected:
    static const int kAllocSize = 4;
    void resize(int newSize);
    int* mElems;
    int mSize;
private:
    // Disallow assignment and pass by value.
    Array(const Array& src);
    Array& operator=(const Array& rhs);
};
```

为了只突出最重要的内容，在此忽略了异常抛出列表，而且没有将此类建立为一个模板。这个接口支持元素的设置和获取。它提供了随机访问保证，客户可以创建数组，并设置第 1、第 100 和第 1000 个元素，而不必担心内存管理。

以下是这些方法的实现：

```
#include "Array.h"

const int Array::kAllocSize;

Array::Array()
{
    mSize = kAllocSize;
    mElems = new int[mSize];
}

Array::~Array()
{
    delete [] mElems;
}
```

```

int Array::getElementAt(int x) const
{
    if (x < 0 || x >= mSize) {
        throw out_of_range("");
    }
    return mElems[x];
}

void Array::setElementAt(int x, int val)
{
    if (x < 0) {
        throw out_of_range("");
    }
    if (x >= mSize) {
        // Allocate kAllocSize past the element the client wants
        resize (x + kAllocSize);
    }
    mElems[x] = val;
}

void Array::resize(int newSize)
{
    int* newElems = new int[newSize]; // Allocate the new array of the new size.

    // The new size is always bigger than the old size.
    for (int i = 0; i < newSize; i++) {
        // Copy the elements from the old array to the new one.
        newElems[i] = mElems[i];
    }
    mSize = newSize; // Store the new size.
    delete [] mElems; // Free the memory for the old array.
    mElems = newElems; // Store the pointer to the new array.
}

```

下面是一个小例子，展示了如何使用这个类：

```

Array arr;
int i;

for (i = 0; i < 10; i++) {
    arr.setElementAt(i, 100);
}

for (i = 0; i < 10; i++) {
    cout << arr.getElementAt(i) << " ";
}

cout << endl;

```

可以看到，不必指出数组需要多大的空间。它会根据提供的元素（即要保存的元素）来分配足够多的空间。不过，使用 `setElementAt()` 和 `getElementAt()` 函数并不方便。要是能如下使用真正的数组下标记法就好了：

```
Array arr;  
int i;  
  
for (i = 0; i < 10; i++) {  
    arr[i] = 100;  
}  
  
for (i = 0; i < 10; i++) {  
    cout << arr[i] << " ";  
}  
cout << endl;
```

这就是为什么要引入重载下标操作符。可以把类中的 `getElementAt()` 和 `setElementAt()` 代之以一个 `operator[]`，如下所示：

```
class Array  
{  
public:  
    Array();  
    ~Array();  
    int& operator[](int x)  
protected:  
    static const int kAllocSize = 4;  
    void resize(int newSize);  
    int* mElems;  
    int mSize;  
private:  
    // Disallow assignment and pass by value.  
    Array(const Array& src);  
    Array& operator=(const Array& rhs);  
};
```

这样，前面在数组中使用数组下标记法的代码就能编译了。`operator[]` 可以取代 `setElementAt()` 和 `getElementAt()`，因为它能返回位于 `x` 处的元素的一个引用。这个引用可以是一个左值，因此可以用于对该元素赋值。以下是这个操作符的实现：

```
int& Array::operator[](int x)  
{  
    if (x < 0) {  
        throw out_of_range("");  
    }  
    if (x >= mSize) {  
        // Allocate kAllocSize past the element the client wants.  
        resize(x + kAllocSize);  
    }  
    return (mElems[x]);  
}
```

`operator[]` 用在赋值语句左边时，赋值实际上修改了 `mElems` 数组中位置 `x` 处的值。

### 16.5.1 利用 `operator[]` 提供只读访问

尽管有时让 `operator[]` 返回一个可以作为左值的元素会很方便，但你不一定希望这样。如果还能够通过返回一个 `const` 值或 `const` 引用，对数组中的元素提供只读访问就好了。理想情况下，要提供两个

1124

```
const int& Array::operator[](int x) const
{
    if (x < 0 || x >= mSize) {
        throw out_of_range("");
    }
    return (mElems[x]);
}
```

14082

个的数，并求其平方和。在程序中，用变量  $sum$  来存放平方和，用变量  $i$  来存放平方数。

```
    printArray(arr, 10);  
    return (0);  
}  
  
void printArray(const Array& arr, int size)  
{  
    for (int i = 0; i < size; i++) {  
        cout << arr[i] << " "; // Calls the const operator[] because arr is a const  
                                // object.  
    }  
    cout << endl;  
}
```

注意, `const operator[]` 只在 `printArray()` 中调用, 因为 `arr` 是 `const`。如果 `arr` 不是 `const`, 就会调用非 `const operator[]`, 尽管结果并未修改。

### 16.5.2 非整数数组索引

还可以编写使用非整数的其他类型作为索引的 `operator[]`。例如, 可以创建一个关联数组 (associative array), 其中用 `string` 而不是用整数作为键。以下是存储 `int` 的一个关联数组类的定义:

```
class AssociativeArray  
{  
public:  
    AssociativeArray();  
    ~AssociativeArray();  
  
    int& operator[](const string& key);  
    const int& operator[](const string& key) const;  
private:  
    // Implementation details omitted  
};
```

我们没有提供这个类的实现, 这作为练习留给读者。你可能还会有兴趣了解 STL `map` 提供了一种关联数组的功能, 可以使用任何可能的类型作为键来使用 `operator[]`。

不能将下标操作符重载为取多个参数。如果想对多个索引提供下标, 可以使用函数调用操作符。

## 16.6 重载函数调用操作符

C++ 允许重载函数调用操作符, 可以写作 `operator()`。如果为类编写了一个 `operator()`, 就可以将该类的对象作为函数指针来使用。只能把这个操作符重载为类中的一个非静态方法。以下是一个简单的类, 其中有一个重载的 `operator()`, 还有一个有相同行为的类方法:

```
class FunctionObject  
{  
public:  
    int operator() (int inParam); // Function-call operator  
    int aMethod(int inParam); // Normal method  
};  
  
//Implementation of overloaded function-call operator  
int FunctionObject::operator() (int inParam)
```



```

{
    return (inParam * inParam);
}

// Implementation of normal method
int FunctionObject::aMethod(int inParam)
{
    return (inParam * inParam);
}

```

以下代码是使用此函数调用操作符的一个例子，并与类方法的正常调用进行对照：

```

int main(int argc, char** argv)
{
    int x = 3, xSquared, xSquaredAgain;
    FunctionObject square;

    xSquared = square(x); // Call the function-call operator
    xSquaredAgain = square.aMethod(x); // Call the normal method
}

```

如果类有函数调用操作符，这个类的对象称为一个函数对象（function object），或简称为 functor。

乍一看，函数调用操作符可能有些奇怪。为什么想要为类编写一个特殊的方法，使该类对象看上去像是函数指针呢？为什么不干脆写一个函数或者类的一个标准方法呢？相对于对象的标准方法来说，函数对象的优点很简单：这些对象有时可以伪装成函数指针。可以将函数对象作为回调函数传递给希望得到函数指针的例程，只要函数指针类型是模板化的。有关详细内容请见第 22 章。

与全局函数相比，函数对象的优点更复杂一些。它有两个主要的好处：

- 对象可以在其数据成员中保留函数调用操作符重复调用之间的信息。例如，函数对象可以用于保存由各函数调用操作符调用收集到的数字之和。
- 可以通过设置数据成员来定制函数的行为。例如，可以编写一个函数对象将函数的一个参数与一个数据成员进行比较。这个数据成员可以是可配置的，从而能定制对象来完成希望的比较。

当然，也可以用全局或静态变量来达到上述好处。不过，函数对象为此提供了一个更简洁的方法。在第 21 章和第 23 章中对 STL 有更多了解之后，函数对象的真正好处会更明显。

通过遵循常规的方法重载规则，可以为类编写任意多个 `operator()`。特别地，不同的 `operator()` 的参数个数或类型必须不同。例如，可以为 `FunctionObject` 对象增加一个 `operator()`，它取一个 `string` 引用为参数：

```

class FunctionObject
{
public:
    int operator() (int inParam);
    void operator() (string& str);
    int aMethod(int inParam);
};

```

函数调用操作符还可以用来为数组的多个索引提供下标。可以简单地编写一个表现为 `operator []` 的 `operator()`，但是允许有多个参数。这种技术惟一的问题是必须使用 `()` 来索引而不是 `[]`，如 `myArray(3, 4) = 6;`



## 16.7 重载解除引用操作符

有三个解除引用操作符可以重载：`*`、`->` 和 `->*`。暂且不管 `->*`（后面再谈），先考虑 `*` 和 `->` 的内置含义。`*` 可以解除一个指针的引用，使你能直接访问其值，而 `->` 是一个 `*`（解除引用）再跟一个（成员选择）的简写。以下代码显示了二者是等价的：

```
SpreadsheetCell* cell1 = new SpreadsheetCell;
(*cell1).set(5); // Dereference plus member selection
cell1->set(5); // Shorthand arrow dereference and member selection together
```

可以对类重载解除引用操作符，从而使该类对象表现为指针。这种功能主要用于实现智能指针，有关内容已经在第 4 章、第 13 章和第 15 章介绍过。这对于迭代器也很有用，STL 就使用这种技术，可以把迭代器认为是奇特的智能指针。第 21 章~第 23 章将会更详细地介绍迭代器，第 25 章提供了智能指针类的一个示例实现。本章将介绍在一个简单智能指针模板类上下文中重载相关操作符的基本做法。

以下是智能指针类实现，在此还没有加入解除引用操作符：

```
template <typename T>
class Pointer
{
public:
    Pointer(T* inPtr);
    ~Pointer();

    // Dereference operators will go here.
protected:
    T* mPtr;
private:
    // Prevent assignment and pass by reference.
    Pointer(const Pointer<T>& src);
    Pointer<T>& operator=(const Pointer<T>& rhs);
};
```

这个智能指针是最简单不过的了。它所做的就是存储一个哑指针，并在撤销对象时删除底层指针。其实现也很简单：构造函数取一个真正（“哑”）指针，把它作为类的惟一数据成员保存起来。析构函数会释放此指针。

```
template <typename T>
Pointer<T>::Pointer(T* inPtr)
{
    mPtr = inPtr;
}

template <typename T>
Pointer<T>::~~Pointer()
{
    delete mPtr;
}
```

可以如下使用智能指针模板：

```
#include "Pointer.h"
#include "SpreadsheetCell.h"
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    Pointer<int> smartInt(new int);

    *smartInt = 5; // Dereference the smart pointer.
    cout << *smartInt << endl;
    Pointer<SpreadsheetCell> smartCell(new SpreadsheetCell);

    smartCell->set(5); // Dereference and member select the set method.
    cout << smartCell->getValue() << endl;

    return (0);
}
```

可以看到，需要为这个类提供 \* 和 -> 的实现。

很少单独编写 operator\* 或 operator->。一般两者都要实现，从而为类提供适当的语义。如果一个智能指针型的对象支持 -> 但不支持 \*，这会很奇怪，反之亦然。

### 16.7.1 实现 operator\*

在解除一个指针的引用时，你会希望访问该指针所指的内存。如果该内存包含一个诸如 int 之类的简单类型，应该能够直接修改它的值。如果内存包含一个更复杂的类型，如一个对象，则应当能利用操作符来访问数据成员或方法。

为了提供上述语义，应当从 operator\* 返回一个变量或对象的引用。在 Pointer 类中，声明和定义如下所示：

```
template <typename T>
class Pointer
{
public:
    Pointer(T* inPtr);
    ~Pointer();
    T& operator*();
    const T& operator*() const;

protected:
    T* mPtr;

private:
    Pointer(const Pointer<T>& src);
    Pointer<T>& operator=(const Pointer<T>& rhs);
};
```

```
template <typename T>
T& Pointer<T>::operator*()
{
    return (*mPtr);
}
```

可以看到, `operator*` 返回底层哑指针所指对象或变量的一个引用。与重载下标操作符类似, 同时提供此方法的 `const` 和非 `const` 版本, 分别返回一个 `const` 引用和一个一般的引用, 这会很有用。 `const` 版本的实现与非 `const` 版本相同, 所以在此不再显示。

### 16.7.2 实现 `operator->`

箭头操作符稍微难一些。应用箭头操作符的结果应当是一个成员或对象的一个方法。不过, 为了如此实现, 必须能够实现与之等价的操作符, 即 `operator*` 后跟有 `operator`。C++ 不允许重载 `operator`, 其理由很充分: 要编写一个原型让你访问所有可能的成员或方法选择, 这是不可能的。类似地, 也不能编写有此语义的 `operator->`。

因此, C++ 把 `operator->` 作为一个特例。考虑以下这行代码:

```
smartCell-> set (5);
```

C++ 会把这行代码翻译为:

```
(smartCell. operator-> ()) -> set (5);
```

可以看到, C++ 实际上对从重载 `operator->` 返回的内容应用了另一个 `operator->`。因此, 必须如下返回一个对象的指针:

```
template <typename T>
class Pointer
{
public:
    Pointer(T* inPtr);
    ~Pointer();
    T& operator*();
    const T& operator*() const;
    T* operator->();
    const T* operator->() const;
protected:
    T* mPtr;
private:
    Pointer(const Pointer<T>& src);
    Pointer<T>& operator=(const Pointer<T>& rhs);
};
```

```
template <typename T>
T* Pointer<T>::operator->()
{
    return (mPtr);
}
```

同样地, 也应该为这个操作符同时编写 `const` 和非 `const` 版本。 `const` 版本的实现与非 `const` 版本完全相同, 因此这里不再显示。

遗憾的是, `operator*` 和 `operator->` 是非对称的, 不过, 只要见过几次, 应该很快就能习惯。

### 16.7.3 到底什么是 `operator->`

第 9 章曾介绍过, 可以管理指向类成员和方法的指针。要对指针解除引用, 必须是在该类对象的上文中。以下是第 9 章中的一个例子。

```
SpreadsheetCell myCell;
double (SpreadsheetCell:: * methodPtr) () const = &SpreadsheetCell:: getValue;
cout<< (myCell. * methodPtr) () << endl;
```

需要说明的是，在此使用了 \* 操作符来解除方法指针的引用，并调用该方法。这与通过指针调用方法的 operator->\* 是等价的，此时已有的是一个对象的指针，而不是对象本身。操作符如下所示：<<

```
SpreadsheetCell* myCell = new SpreadsheetCell();
double (SpreadsheetCell::*methodPtr) () const = &SpreadsheetCell::getValue;
cout << (myCell->*methodPtr) () << endl;
```

C++ 不允许重载 operator\* (这与不能重载 operator 是一样的)，不过重载 operator->\* 则是允许的。但是，重载 operator->\* 很困难，而且如果大多数 C++ 程序员都不知道可以通过指针来访问方法和成员，这么麻烦可能是不值得的。标准库中的 auto\_ptr 模板就没有重载 operator->\*。

## 16.8 编写转换操作符

再回到 SpreadsheetCell 例子上来，考虑以下两行代码：

```
SpreadsheetCell cell1;
string s1 = cell1; // DOES NOT COMPILE!
```

SpreadsheetCell 包含一个字符串表示，因此将其赋值给一个 string 变量看上去是合理的。但是这样并不可行。编译器会指出它不知道怎么把一个 SpreadsheetCell 转换为一个 string。你可能想强制编译器按你的预想如下去做：

```
string s1 = (string) cell1; // STILL DOES NOT COMPILE!
```

首先，上面这行代码还是不能编译，因为编译器现在还是不知道怎么把 SpreadsheetCell 转换为 string。它已经从前面的第一行知道了你想做什么，如果它能做早就做了。其次，一般来讲，在程序中加入莫名其妙的强制转换不是一个好做法。即使编译器允许这个强制转换通过编译，在运行时也可能做得并不正确。例如，它可能会把表示对象的位解释为一个 string。

如果想要允许这种赋值，就必须告诉编译器如何完成这种操作。具体地，可以编写一个转换操作符将 SpreadsheetCell 转换为 string。其原型如下所示：

```
class SpreadsheetCell
{
public:
    // Omitted for brevity
    operator string() const;
    // Omitted for brevity
};
```

函数名为 operator string。它没有返回类型，因为返回类型已经由操作符名指定为 string。这是一个 const 函数，因为它没有改变调用此操作的对象。确实，乍一看有些奇怪，但你会习惯的。其实现如下所示：

```
SpreadsheetCell::operator string() const
{
    return (mString);
}
```

要编写一个转换操作符将 SpreadsheetCell 转换为 string，所要做的只有这么多。现在编译器可以接受这行代码，而且在运行时能够正确地工作。

```
SpreadsheetCell cell1;  
string s1 = cell1; // Works as expected
```

可以为任何类型编写有同样语法的转换操作符。例如，以下是从 SpreadsheetCell 转换到 double 转换操作符的原型：

```
class SpreadsheetCell  
{  
public:  
    // Omitted for brevity  
    operator string() const;  
    operator double() const;  
    // Omitted for brevity  
};
```

其实现如下所示：

```
SpreadsheetCell::operator double() const  
{  
    return (mValue);  
}
```

现在可以写出如下的代码：

```
SpreadsheetCell cell1;  
double d2 = cell1;
```

### 16.8.1 转换操作符的二义性问题

遗憾的是，从 SpreadsheetCell 对象转换到 double 时，为此编写转换操作符会带来一个二义性问题。考虑下面这两行代码：

```
SpreadsheetCell cell1;  
double d1 = cell1 + 3.3; // DOES NOT COMPILE IF YOU DEFINE operator double()
```

这两行代码无法编译。在写 operator double() 之前这些代码本来能正常工作的，那么现在出了什么问题呢？问题在于，编译器现在不知道该用 operator double() 将 cell1 转换为一个 double，并完成 double 加法运算，还是要用 double 版本的构造函数将 3.3 转换为一个 SpreadsheetCell，并完成 SpreadsheetCell 加法操作。在编写 operator double() 之前，编译器只有一个选择：只能用 double 版本的构造函数将 3.3 转换为一个 SpreadsheetCell，并完成 SpreadsheetCell 加法操作。不过，现在这两个选择都是可以的。而编译器不会做出选择，因为你可能不喜欢它的选择，所以它干脆拒绝做任何选择。

解决这个难题的一般方案是置当前构造函数为 explicit，这样就能避免使用此构造函数的自动转换。遗憾的是，我们不希望构造函数为 explicit，因为一般我们都希望 double 能自动转换为 SpreadsheetCell，这在第 9 章已经解释过。在这种情况下，可能更好的做法是不要为 SpreadsheetCell 类编写 double 转换操作符。



### 16.8.2 布尔表达式的转换

有时如果能在布尔表达式中使用对象会很有用。例如，程序员通常在条件语句中如下使用指针：

```
if (ptr != NULL) {
    // Perform some dereferencing action.
}
```

有时会写出简写条件，如下所示：

```
if (ptr) {
    // Perform some dereferencing action.
}
```

在另外一些情况下，你还会看到如下的代码：

```
if (!ptr) {
    // Do something.
}
```

就目前而言，以上带 Pointer 智能指针类（如前定义）的表达式都无法通过编译。不过，你可以为类增加一个转换操作符，将其转换为一个指针类型。这样一来，无论是到 NULL 的转换，还是 if 语句中单独出现的对象都会触发到指针类型的转换。转换操作符通常的指针类型为 void\*。以下是修改后的 Pointer 类：

```
template <typename T>
class Pointer
{
public:
    Pointer(T* inPtr);
    ~Pointer();
    T& operator*();
    const T& operator*() const;
    T* operator->();
    const T* operator->() const;
    operator void*() const { return mPtr; }
protected:
    T* mPtr;
private:
    Pointer(const Pointer<T>& src);
    Pointer<T>& operator=(const Pointer<T>& rhs);
};
```

现在以下语句都能成功编译，并能完成你希望的工作。

```
Pointer<SpreadsheetCell> smartCell(new SpreadsheetCell);
smartCell->set(5);
if (smartCell != NULL) {
    cout << "not NULL!\n";
}
if (smartCell) {
    cout << "not NULL!\n";
}
if (!smartCell) {
    cout << "NULL!\n";
}
```



还有一种做法是重载 `operator bool` 而不是 `operator void*`。毕竟，你是在布尔表达式中使用对象。为什么不把它直接转换为一个 `bool` 呢？可以如下编写 `Pointer` 类：

```
template <typename T>
class Pointer
{
public:
    Pointer(T* inPtr);
    ~Pointer();
    T& operator*();
    const T& operator*() const;
    T* operator->();
    const T* operator->() const;
    operator bool() const { return (mPtr != NULL); }
protected:
    T* mPtr;
private:
    Pointer(const Pointer<T>& src);
    Pointer<T>& operator=(const Pointer<T>& rhs);
};
```

上述三个测试仍能工作，不过显式转换为 `NULL` 可能会使编译器产生警告。这个技术对于不表示指针而且转换到指针类型并没有意义的对象看上去特别合适。遗憾的是，增加一个到 `bool` 的转换操作符会带来一些预计不到的后果。C++ 会在机会出现的时候应用“提升”原则将 `bool` 悄悄地转换为 `int`。因此，如果有以上转换操作符，下面的代码就能编译通过并运行。

```
Pointer<SpreadsheetCell> smartCell(new SpreadsheetCell);
int i = smartCell; // Converts smartCell Pointer to bool to int.
```

这通常并非你的本意。因此，许多程序员倾向于实现 `operator void*` 而不是 `operator bool`。实际上，可以回忆第 14 章给出的以下流使用：

```
ifstream istr;
int temp;
// Open istr
while (istr >> temp) {
    // Process temp
}
```

为了能在布尔表达式中使用流对象，而且禁止不需要的提升至 `int`，`basic_ios` 类定义了 `operator void*` 而不是 `operator bool`。

第三种方法是实现 `operator!`，并要求类的客户只使用取反转换，如：

```
if (!smartCell) {
    cout << " NULL\n";
}
```

可以看到，重载操作符时需要有所设计。对于重载哪些操作符，你的决定将直接影响着客户如何使用你的类。

## 16.9 重载内存分配和撤销操作符

C++ 允许在程序中重新定义内存分配和撤销的工作方式。可以在全局层次和类层次提供此定制。如

果你要考虑性能，而且希望提供比默认方式更高效的内存管理，这种能力就最为有用。例如，并非每次需要内存时都要求完成默认的 C++ 内存分配，可以编写一个内存池分配程序，它能重用固定大小的内存块。这一节将解释内存分配和撤销例程的细节，并说明如何进行定制。利用这些工具，如果有需求就能编写一个内存池了。

除非你很了解内存分配策略，否则重载内存分配例程所带来的麻烦往往大于它的价值。不要只是因为听上去像是一个不错的想法就草率地进行重载。只有当你确实有性能或空间需求而且有必要的知识储备时，才可以这么做。

### 16.9.1 new 和 delete 究竟如何工作

C++ 中最麻烦的一个方面就是 new 和 delete 的细节。请考虑以下这行代码：

```
SpreadsheetCell* cell = new SpreadsheetCell();
```

new SpreadsheetCell() 称为 new 表达式 (new expression)。它会做两件事。首先，通过调用 operator new 会为 SpreadsheetCell 对象分配空间。其次，它会调用对象的构造函数，只有当构造函数完成之后，它才会返回指针。

delete 的作用是类似的，考虑以下这行代码：

```
delete cell;
```

这行代码称为 delete 表达式 (delete expression)。它首先调用 cell 的析构函数，然后调用 operator delete 来释放内存。

使用关键字 new 分配内存时，并没有直接调用 operator new。在使用关键字 delete 释放内存时，也不是直接调用 operator delete。

可以重载 operator new 和 operator delete 来控制内存分配和撤销，不过不能重载 new 表达式或 delete 表达式。因此，可以定制具体的内存分配和撤销，但不能定制对构造函数和析构函数的调用。

#### new 表达式和 operator new

new 表达式有 6 种不同的形式，每一种都有一个相应的 operator new。

在第 13 章和第 15 章已经见过前面的 4 种 new 表达式：new、new []、nothrow new 和 nothrow new []。这 4 种表达式的相应 operator new 都定义在头文件 <new> 中，在此将它们再列出来：

```
void* operator new(size_t size) throw(bad_alloc); // For new
void* operator new[](size_t size) throw(bad_alloc); // For new[]
void* operator new(size_t size, const nothrow_t&) throw(); // For nothrow new
void* operator new[](size_t size, const nothrow_t&) throw(); // For nothrow new[]
```

第 5 种和第 6 种形式的 new 称为定位 new (placement new) (包括单个和数组形式)。利用这两种形式可以在预存的内存中构造一个对象，如下所示：

```
void* ptr = allocateMemorySomehow();
SpreadsheetCell* cell = new (ptr) SpreadsheetCell();
```

这个特性有些含糊，不过重要的是要知道这个特性的存在。如果你想实现内存池，以便重用内存而无需在两次使用之间进行释放，这就很方便。相应的 operator new 如下所示：

```
void* operator new(size_t size, void* p) throw();  
void* operator new[](size_t size, void* p) throw();
```

### Delete 表达式和 operator delete

可以调用的 delete 表达式只有两种不同形式：delete 和 delete []；在此没有 nothrow 或定位形式。不过，确实存在 6 种不同形式的 operator delete。为什么二者不对称呢？除了以上可调用的两种形式外，另外 4 种 nothrow 或定位形式仅在从构造函数抛出一个异常时才使用。在这种情况下，构造函数调用前可能使用了某种形式的 operator new 来分配内存，此时则要调用与该 operator new 相匹配的 operator delete。不过，如果正常地删除一个指针，delete 会调用 operator delete 或 operator delete []（而不是 nothrow 或定位形式）。

实际上，这也没什么关系，delete 不会抛出异常，因此，operator delete 的 nothrow 版本是多余的，而且定位形式的 delete 应当是一个不做任何工作的操作（no-op）。（内存不是在定位 operator new 中分配的，因此没有什么要释放）。以下是各种形式 operator delete 的原型：

```
void operator delete(void* ptr) throw();  
void operator delete[](void* ptr) throw();  
void operator delete(void* ptr, const nothrow_t&) throw();  
void operator delete[](void* ptr, const nothrow_t&) throw();  
void operator delete(void* p, void*) throw();  
void operator delete[](void* p, void*) throw();
```

### 16.9.2 重载 operator new 和 operator delete

如果需要，确实可以替换全局的 operator new 和 operator delete 例程。对于程序中的每个 new 表达式和 delete 表达式都会调用这些函数，除非在各个类中有更特定的例程。不过，按 Bjarne Stroustrup 的话说，“……如果没有足够的胆量，就不要替换全局 operator new() 和 operator delete()。”（《C++ 编程语言》，第 3 版）。我们也不建议这么做！

如果没听从我们的建议，还是决定替换全局 operator new，要记住，不能在这个操作符中放任何调用 new 的代码，否则会导致一个无限循环。例如，不能用 cout 向控制台写一条消息。

一种更有用的技术是为特定的类重载 operator new 和 operator delete。这些重载操作符只会在分配和撤销该类对象时才会调用。以下是一个类的例子，它重载了 4 种非定位形式的 operator new 和 operator delete：

```
#include <new>  
using namespace std;  
  
class MemoryDemo  
{  
public:  
    MemoryDemo() {}  
    ~MemoryDemo() {}  
  
    void* operator new(size_t size) throw(bad_alloc);  
    void operator delete(void* ptr) throw();  
  
    void* operator new[](size_t size) throw(bad_alloc);  
    void operator delete[](void* ptr) throw();
```

```

void* operator new(size_t size, const nothrow_t&) throw();
void operator delete(void* ptr, const nothrow_t&) throw();

void* operator new[](size_t size, const nothrow_t&) throw();
void operator delete[](void* ptr, const nothrow_t&) throw();
};

```

以下是这些操作符的简单实现，它们只是把参数传递给全局版本的操作符调用。注意，nothrow 实际上是类型为 nothrow\_t 的一个变量。

```

#include "MemoryDemo.h"
#include <iostream>
using namespace std;
void* MemoryDemo::operator new(size_t size) throw(bad_alloc)
{
    cout << "operator new\n";
    return (::operator new(size));
}

void MemoryDemo::operator delete(void* ptr) throw()
{
    cout << "operator delete\n";
    ::operator delete(ptr);
}

void* MemoryDemo::operator new[](size_t size) throw(bad_alloc)
{
    cout << "operator new[]\n";
    return (::operator new[](size));
}

void MemoryDemo::operator delete[](void* ptr) throw()
{
    cout << "operator delete[]\n";
    ::operator delete[](ptr);
}

void* MemoryDemo::operator new(size_t size, const nothrow_t&) throw()
{
    cout << "operator new nothrow\n";
    return (::operator new(size, nothrow));
}

void MemoryDemo::operator delete(void* ptr, const nothrow_t&) throw()
{
    cout << "operator delete nothrow\n";
    ::operator delete[](ptr, nothrow);
}

void* MemoryDemo::operator new[](size_t size, const nothrow_t&) throw()
{
    cout << "operator new[] nothrow\n";
    return (::operator new[](size, nothrow));
}

```

```
void MemoryDemo::operator delete[](void* ptr, const nothrow_t&) throw()
{
    cout << "operator delete[] nothrow\n";
    ::operator delete[](ptr, nothrow);
}
```

以下代码显示了可以采用多种方式分配和撤销该类对象：

```
#include "MemoryDemo.h"

int main(int argc, char** argv)
{
    MemoryDemo* mem = new MemoryDemo();
    delete mem;

    mem = new MemoryDemo[10];
    delete [] mem;

    mem = new (nothrow) MemoryDemo();
    delete mem;

    mem = new (nothrow) MemoryDemo[10];
    delete [] mem;

    return (0);
}
```

下面是运行程序的输出：

```
operator new
operator delete
operator new[]
operator delete[]
operator new nothrow
operator delete
operator new[] nothrow
operator delete[]
```

operator new 和 operator delete 的这些实现显然很简单，而且没有什么太大用途。这只是想让你了解有关的语法，如果想实现更复杂的版本时就能由此获得帮助。

在重载 operator new 时，还要重载相应形式的 operator delete。否则，会按你指定的方式分配内存，但释放时则按照内置语义，这可能存在不兼容（不一致）。

重载所有 4 种形式的 operator new 看上去有些过分。不过，这样做通常是一个好的想法，这样可以避免内存分配的不一致性。如果不想提供实现，可以将函数声明为 protected 或 private，以避免别人使用它。

重载所有形式的 operator new，或者提供不带实现的 private 声明来避免使用此操作符。

### 16.9.3 重载带额外参数的 operator new 和 operator delete

除了重载标准形式的 operator new 之外，还可以基于额外参数编写自己的版本。例如，以下是 Mem-



oryDemo 类，在此显示了另一个带有一个额外整型参数的 operator new:

```
#include <new>
using namespace std;

class MemoryDemo
{
public:
    MemoryDemo();
    ~MemoryDemo();
    void* operator new(size_t size) throw(bad_alloc);
    void operator delete(void* ptr) throw();
    void* operator new[](size_t size) throw(bad_alloc);
    void operator delete[](void* ptr) throw();
    void* operator new(size_t size, const nothrow_t&) throw();
    void operator delete(void* ptr, const nothrow_t&) throw();
    void* operator new[](size_t size, const nothrow_t&) throw();
    void operator delete[](void* ptr, const nothrow_t&) throw();
    void* operator new(size_t size, int extra) throw(bad_alloc);
};

void* MemoryDemo::operator new(size_t size, int extra) throw(bad_alloc)
{
    cout << "operator new with extra int arg\n";
    return (::operator new(size));
}
```

在编写一个带额外参数的重载 operator new 时，编译器会自动地支持相应的 new 表达式。因此，现在可以编写以下的代码：

```
int x = 5;
MemoryDemo* memp = new(5) MemoryDemo();
delete memp;
```

new 的额外参数按函数调用语法传递（类似于 nothrow new）。这些额外参数对于向内存分配例程传递各种标志或计数器很有用。

不能为 operator delete 增加任意的额外参数。不过，还有一种形式的 operator delete 可以给出应当释放的内存大小以及指针。只需为带有一个额外大小参数的 operator delete 声明原型。

如果类声明了两个基本相同的 operator delete 版本，只是其中一个要取大小参数，另一个无此参数，那么没有大小参数的版本总是会得到调用。如果希望使用带大小参数的版本，就只能编写该版本（而不能再编写另外的那个版本）。

可以把 operator delete 换成另一个版本，对任何版本的 operator delete 独立地取一个大小。以下是修改后的 MemoryDemo 类定义，其中第一个 operator delete 修改为要取所删除内存的大小。

```
#include <new>
using namespace std;
class MemoryDemo
{
public:
    MemoryDemo();
    ~MemoryDemo();
    void* operator new(size_t size) throw(bad_alloc);
```



```
void operator delete(void* ptr, size_t size) throw();  
void* operator new[](size_t size) throw(bad_alloc);  
void operator delete[](void* ptr) throw();  
void* operator new(size_t size, const nothrow_t&) throw();  
void operator delete(void*ptr, const nothrow_t&) throw();  
void* operator new[](size_t size, const nothrow_t&) throw();  
void operator delete[](void* ptr, const nothrow_t&) throw();  
void* operator new(size_t size, int extra) throw(bad_alloc);  
};
```

这个 operator delete 的实现调用了不带大小参数的全局 operator delete，因为任何全局 operator delete 都不取大小。

```
void MemoryDemo::operator delete(void* ptr, size_t size) throw()  
{  
    cout << "operator delete with size\n";  
    ::operator delete(ptr);  
}
```

如果你要为类编写一个复杂的内存分配和撤销机制，这个功能才有用。

## 16.10 小结

这一章总结了操作符重载的有关原理，并提供了重载各种操作符的具体示例和解释。希望通过这一章的介绍，你不再害怕操作符重载的复杂语法，而且能够尽享其提供的强大功能。本书后面的章节将利用操作符重载来提供抽象，包括第 23 章中的迭代器和第 25 章中的智能指针。

# 第17章 编写高效的 C++ 程序

无论是什么应用领域，程序的效率都相当重要。如果你的产品要在市场上与其他产品一决高低，速度可能是一个主要的决定因素：给你一个比较慢的程序和一个比较快的程序，你会选择哪一个呢？如果一个操作系统要花两个星期才能启动，没有人会买这样的操作系统，除非别无选择。即使你不想出售产品，它们总会有用户。如果这些用户要花大量时间等待你的程序完成任务，他们肯定不会乐意。

既然已经了解了专业 C++ 设计和编码的概念，而且也知道了 C++ 语言提供的一些更复杂的工具，下面可以考虑程序的性能了。编写高效的程序需要在设计层次上做考虑，并在实现层次上考虑细节。尽管这一章在本书中的位置不算靠前，但要记住，一定要在程序生命期的最开始就考虑性能。

本章首先会提供与软件相关的“效率”和“性能”的实际定义，并介绍可以在两个层次上提高程序的效率。本章还讨论了两大类应用，随后提供了编写高效程序的具体策略，包括语言级优化和设计级原则。最后，本章会对测评工具做深入的讨论。

## 17.1 性能和效率概述

在进一步深入到细节之前，先对本书中所用的“性能”和“效率”这两个术语给出定义会比较好。程序的性能（performance）可以指许多方面，如速度、内存使用、磁盘存取和网络使用。本章主要强调速度性能。谈到程序的效率（高效率）时，是指没有浪费地顺利运行。高效的程序会在给定环境中尽可能快地完成任务。不快的程序也可能是高效的，因为有的应用领域本身的特点就使得相应程序不可能很快地执行。

高效或高性能程序会尽可能快地运行，来完成特定任务。

请注意本章的标题：“编写高效的 C++ 程序”，这是指要编写能高效运行的程序，而不是高效地编写程序。也就是说，通过阅读本章，学会如何节省用户的时间，而不是节省你自己的时间！

### 17.1.1 实现高效的两种方法

编写高效程序的传统方法是优化（optimizing），或改善既有代码的性能。这个技术通常只涉及语言级效率（language-level efficiency）：做一些特定的、独立的代码修改，如按引用传递对象而不是按值传递。这种方法作用很有限。如果想编写真正高性能的应用，就必须从设计一开始就考虑效率问题。设计级效率（design-level efficiency）包括选择高效算法，避免不必要的步骤和计算，而且要选择适当的设计优化。

### 17.1.2 两类程序

如前所述，效率对所有应用领域都很重要。另外，一小部分程序则需要极高的效率，如系统级软件、嵌入式系统、计算量极大的应用和实时游戏等。大多数程序都没有这么高的要求。除非要编写上述高性

能应用，否则可能不需要对 C++ 代码的速度锱铢必较。可以把这看作是造普通型家用轿车和造赛车之间的区别。每辆车都应该相当高效，但是赛车则要求极高的性能才行。如果家用轿车再快也不会超过每小时 70 英里，你可能不会想着浪费时间去优化这样一辆车。

### 17.1.3 C++ 是一种低效语言吗

C 程序员通常拒绝使用 C++ 来编写高性能应用。他们声称 C++ 本质上比 C 或类似的过程性语言效率要低一些。从表面看来，这种看法是有道理的，C++ 包括一些高级构造（construct），如异常和虚方法，这些构造都相当慢。不过，这种观点也存在一些问题。

首先，不能忽略编译器的影响。在讨论一种语言的效率时，必须把语言本身的性能与编译器对其优化的效力分开来谈。应该记得，你编写的 C 或 C++ 代码并不是计算机实际执行的代码。编译器首先会把你的代码翻译成机器语言，在此过程中会应用优化。这说明，不能只是运行 C 和 C++ 程序的基准测评（benchmark）来比较结果。倘若只比较基准测评结果，那么比较的只是编译器对语言的优化，而不是语言本身。C++ 编译器可以对语言中的许多高级构造“优化”，生成的机器码可能与相应 C 程序所生成的机器码是相当的。

不过，对 C++ 还是存在批评，认为 C++ 的一些特性是不能优化的。例如，在第 10 章中曾解释过，虚方法需要有一个 vtable，而且在运行时还会带来另外的一个间接层，这就造成虚方法要比常规的非虚函数调用慢很多。不过，再仔细考虑一下，这种观点还有待商榷。虚方法调用提供的不只是一个函数调用，它们还使你能够在运行时选择要调用哪个函数。相应的非虚函数调用可能需要一个条件语句来决定要调用哪个函数。如果不需要这些额外的语义，就可以使用一个非虚函数（不过，出于安全和风格方面的原因，建议还是不要采用非虚方法）。C++ 语言中有一个一般性的设计原则：“如果不用它，就不必为之付出。”如果不使用虚方法，就不用付出使用虚方法的性能代价。因此，C++ 中的非虚函数调用与 C 中的函数调用在性能方面是完全相同的。

不过，这些批评在某种意义上也确实属实，C++ 的某些方面导致了很容易在语言级编写低效的代码。不加区分地使用异常和虚函数会让程序慢下来。不过，与 C++ 对算法和整体设计所带来的好处相比，这个问题并不算什么。C++ 的高级构造使你能够编写更简洁的程序，它们在设计层次上更高效、更易于维护，而且可以避免堆积一些不必要的代码和死代码。

最后，本书的两位作者都曾使用 C++ 编写过成功的系统级软件，这些软件中对性能的要求极高。我们相信，通过选择 C++ 而不是另外的一种过程性语言，你能够更好地开发，提供更高的性能和维护性。

## 17.2 语言级效率

许多书、文章和程序员都会花大量时间劝你对代码应用高级优化。这些技巧很重要，在许多情况下确实可以加快程序运行。不过，与程序中的整体设计和算法选择相比，这些优化的重要性就可谓小巫见大巫了。例如，可以把要传递的内容都按引用传递，但是如果你不必要地完成了两倍的磁盘写操作，尽管采用了按引用传递这种优化技术，也不会让你的程序变快多少。另外，这样一来，还很容易陷入到引用和指针当中，而忘记了整个全景图。

另外，好的优化编译器可以自动完成某些语言级优化。花时间自己来优化某个特定方面之前，请查看编译器文档，了解有关的详细内容。

在这本书中，我们力图提供一些合理的策略。所以，在此将包括我们认为最有用的语言级优化。这里所列的优化技术并不完备，但能作为一个很好的起点，可以以此起步开始优化代码。不过，一定要阅读并具体实践本章后面介绍的设计级效率建议。

明智地应用语言级优化。

### 17.2.1 高效地处理对象

C++在后台做了很多工作，特别是在对象方面。一定要当心所写代码的性能影响。如果遵循一些简单的原则，代码就会变得高效得多。

#### 传引用

这条规则在这本书的许多地方都谈到了，不过这里还是有必要重申一下。

对象应当很少按值传递给函数或方法。

传值（Pass-by-value，或按值传递）会带来复制开销，而采用传引用（pass-by-reference，按引用传递）就可以避免这些开销。有时很难记住这条规则，原因是从表面上看按值传递好像也没有什么问题。请考虑一个表示人的类，如下所示：

```
class Person
{
public:
    Person();
    Person(const string& inFirstName, const string& inLastName, int inAge);
    string getFirstName() { return firstName; }
    string getLastName() { return lastName; }
    int getAge() { return age; }

private:
    string firstName, lastName;
    int age;
};
```

可以编写一个函数，它取一个 Person 对象作为参数，如下所示：

```
void processPerson(Person p)
{
    // Process the person,
}
```

可以如下调用这个函数：

```
Person me("Nicholas", "Solter", 28);
processPerson(me);
```

与编写以下函数相比，上面的函数定义似乎也没有多加更多代码：

```
void processPerson(const Person& p)
{
    // Process the person.
}
```

对函数的调用都是一样的。但是，请考虑一下，在第一个版本的函数中按值传递时会发生什么情况。为了初始化 processPerson() 的 p 参数，必须利用 Person 的复制构造函数调用来复制 me。即使没有为 Person 类编写复制构造函数，编译器也会生成一个复制构造函数来复制各个数据成员。不过，Person 类

中有两个数据成员都是 string，它们本身也是带复制构造函数的对象。因此，这两个数据成员的复制构造函数也会被调用。如果 processPerson() 按引用取 p 参数，就不会带来这么多复制开销了。因此，在这个例子中，进入函数时，通过传引用可以避免 3 个函数调用（译者注：即一个 Person 类复制构造函数调用和两个 string 复制构造函数调用）。

还不仅如此。要记住，第一个版本的 processPerson() 中，参数 p 对于 processPerson() 函数来说是一个局部变量，因此函数退出时必须撤销。这个撤销就要求调用 Person 的析构函数。因为没有编写析构函数，默认的析构函数只是会调用所有数据成员的析构函数。string 有析构函数，因此退出这个函数时（如果按值传递）会带来 3 个析构函数调用。如果按引用传递 Person 对象，这些调用都不会存在。

总的说来，如果一个函数必须修改一个对象，可以简单地按引用传递对象。如果函数不会修改对象，则应按 const 引用来传递，如前例所示。有关引用和 const 的详细内容请参见第 12 章。

### 按引用返回

应当按引用向函数传递对象，与此类似，也应当从函数按引用返回对象，以避免不必要的复制对象。遗憾的是，有时不可能按引用返回对象，例如在编写重载 operator+ 和其他类似操作符时便是如此。不能返回一个局部对象的引用或指针，因为局部对象会在函数退出时撤销。

### 按引用捕获异常

在第 15 章中已经指出，应当按引用捕获异常，以避免额外的复制。如本节后面所述，异常在性能方面是重量级的，因此在异常方面做小幅改进也会大大改善效率。

### 避免创建临时对象

在许多情况下编译器会创建临时的匿名对象。回忆第 9 章的介绍，为一个类编写了一个全局 operator+ 之后，可以将该类对象与其他类型相加，只要这些类型可以转换为该类对象即可。例如，SpreadsheetCell 类定义中的一部分如下所示：

```
class SpreadsheetCell
{
public:
    // Other constructors omitted for brevity
    SpreadsheetCell(double initialValue);

    friend const SpreadsheetCell operator+(const SpreadsheetCell& lhs,
        const SpreadsheetCell& rhs);

    // Remainder omitted for brevity
};
```

由于有取 double 的构造函数，因此可以写出以下的代码：

```
SpreadsheetCell myCell(4), aThirdCell;

aThirdCell = myCell + 5.6;
aThirdCell = myCell + 4;
```

第一行完成加法的代码根据参数 5.6 构造了一个临时的 SpreadsheetCell 对象，然后调用 operator+，并以 myCell 和这个临时对象作为参数。结果保存在 aThirdCell 中。第二行完成加法的代码也做了同样的工作，只不过 4 要强制转换为一个 double，这样才能调用 SpreadsheetCell 的 double 版本构造函数。

上面这个例子的重点是：编译器会为每行加法代码生成必要的代码来创建一个额外的匿名 SpreadsheetCell 对象。这个对象必须用构造函数和析构函数进行构造和撤销。如果你还心存怀疑，可以尝试在

构造函数和析构函数中插入一些 `cout` 语句，查看打印结果就会明白了。

一般来说，只要代码将某种类型的变量转换为另外一种类型，从而用在一个更大的表达式中，此时编译器就会构造一个临时的对象。这条规则主要适用于函数调用。例如，假设你编写了一个签名的函数：

```
void doSomething(const SpreadsheetCell& s);
```

可以这样来调用：

```
doSomething(5.56);
```

编译器使用 `double` 版本的构造函数从 `5.56` 构造了一个临时的 `SpreadsheetCell` 对象，并把这个对象传递给 `doSomething()`。需要说明，如果去掉 `s` 参数上的 `const`，就不能基于一个常量来调用 `doSomething()` 了，因为你必须为这个函数传递一个变量。临时对象只能作为 `const` 引用的目标，而不能作为非 `const` 引用的目标。

一般应该尽量避免让编译器构造临时对象的情况。尽管在某些情况下这是无法避免的，但是至少应该知道存在这个“特性”，这样在看到性能和测评分析结果时就不至于感到奇怪了。

#### 返回值优化

按值返回对象的函数可能导致创建一个临时对象。继续看 `Person` 例子，请考虑以下函数：

```
Person createPerson()  
{  
    Person newP;  
    return (newP);  
}
```

如果如下调用（假设这里的 `operator<<` 是为 `Person` 类实现的重载操作符）：

```
cout << createPerson();
```

即使这个调用没有在任何位置上保存 `createPerson()` 的结果，其结果也肯定会存储在某个地方，以便传递给 `operator<<` 调用。为了生成代码来实现这种行为，编译器会创建一个临时变量，在其中存储 `createPerson()` 返回的 `Person` 对象。

即使这个函数的结果未在任何地方使用，编译器还是会生成代码来创建这个临时对象。例如，假设有以下代码：

```
createPerson();
```

编译器可能会生成代码为返回值创建一个临时对象，尽管这个对象根本不会用到。

不过，一般不必担心这个问题，因为编译器会把大多数情况下的临时变量予以优化。这种优化称为返回值优化（`return value optimization`）。

### 17.2.2 不要过度使用高开销的语言特性

从执行速度方面看，许多 C++ 特性的开销都很大，异常、虚方法和 RTTI（运行时类型识别）更是首当其冲。如果把效率摆在重要的位置上，就应该考虑避免这些特性。遗憾的是，即使在程序中没有显式地使用异常和 RTTI 特性，这些特性也会带来性能开销。仅仅是因为可能会用到这些特性，为了对此提供支持，都会在执行中增加额外的步骤。因此，许多编译器允许指定编译程序的方式，即编译程序时可以根本不支持这些特性。例如，考虑以下的简单程序，其中使用了异常和 RTTI：



```
// test.cpp
#include <iostream>
#include <exception>
using namespace std;

class base
{
public:
    base() {}
    virtual ~base() {}
};

class derived : public base {};

int main(int argc, char** argv)
{
    base* b = new derived();
    derived* d = dynamic_cast<derived*>(b); // Use RTTI.
    if (d == NULL) {
        throw exception(); // Use exceptions.
    }
    return (0);
}
```

在 Linux 上使用 g++ 3.2.2，可以采用以下方式编译这个程序：

```
>g++ test.cpp
```

如果为 g++ 指定标志来禁用异常，可以如下编译：

```
>g++ -fno-exceptions test.cpp
test.cpp: In function 'int main (int, char**)':
test.cpp:20: exception handling disabled, use -fexceptions to enable
```

奇怪的是，如果为 g++ 指定标志禁用 RTTI，编译器还会成功地编译这个程序，尽管程序中显然用到了 `dynamic_cast`：

```
>g++ -fno-rtti test.cpp
>
```

不过，由于使用了 RTTI，在运行时就会出错，会导致程序产生一个段冲突。

请参考编译器文档，了解禁用这些特性的相应标志。

禁用对语言特性的支持可能有风险。你不知道什么时候一个第三方库会突然抛出一个异常，或者依赖于 RTTI 才能实现正确的行为。因此，只有当你确信所有代码和使用的库代码都不需要这些特性，而且你编写的应用对性能的要求极高，此时才可以禁用对异常和 RTTI 的支持。

### 17.2.3 使用内联方法和函数

如第 9 章所述，内联方法和函数的代码会直接插入到代码中调用该方法或函数的位置上，这样就避免了函数调用的开销。如果觉得某些函数或方法能够做此优化，就应该把所有这些满足要求的函数和方法都标以 `inline`。不过，要记住，程序员的内联请求只是对编译器提出一个建议。即使你想内联某个函数，编译器也可以拒绝内联该函数。

另一方面，一些编译器会在完成优化的步骤中对适当的函数和方法进行内联，即使这些函数未标以 inline 关键字。因此，在花大量时间决定哪些函数要内联之前，应该先看看编译器文档。

### 17.3 设计级效率

程序中的设计选择对性能的影响远远超过了语言级选择（如传引用）的影响。例如，如果你为应用中的一个基本任务选择了一个运行时间为  $O(n^2)$  的算法，而不是运行时间为  $O(n)$  的更简单的算法，可能会不必要地完成更多的操作，即操作次数会是实际所需操作次数的平方。用具体数字来说明，一个任务使用一个  $O(n^2)$  的算法要完成 1000 000 次操作，倘若采用一个  $O(n)$  的算法则只需完成 1000 次操作。即使在语言级对该操作做了优化，但是要完成 1000 000 次操作，而更好的算法只完成 1000 次操作，仅凭这一点就会使程序效率非常低下。不过，要记住，大  $O$  记法忽略了一些常量因素，因此这不一定是最合理的指导原则。不管怎么说，都应该仔细地选择算法。请参考本书第一部分（特别是第 4 章），其中详细讨论了算法设计的选择。

除了算法选择外，设计级效率还包括一些特殊的技巧。本节余下的内容就是要介绍优化程序的 3 个设计技术：缓存、对象池和线程池。

#### 17.3.1 尽可能缓存

缓存（Caching）是指存储有关内容以便将来使用，从而避免获取或再次计算。你应该已经很了解缓存在计算机硬件中的使用，所以可能对这个原则并不陌生。现代计算机处理器都备有内存缓存，可以在一个位置中存储最近的经常访问的内存值，访问此缓存比访问主存更快。大多数要访问的内存位置都会在很短的时间内被访问不只一次，因此硬件级的缓存可以大大加快计算速度。

软件中的缓存也沿用同样的方法。如果一个任务或计算特别慢，就应当确保自己没有不必要地多次完成这个任务。在第一次完成这个任务时将结果存储在内存中，这样就可以在以后需要时使用这些结果。下面列出的任务往往很慢：

- 磁盘存取。在程序中应当避免多次打开和读取同一个文件。如果有足够的可用内存，而且如果会经常地存取这个文件，就应当把文件内容保存在 RAM 中。
- 网络通信。需要通过网络通信时，程序就会遭遇变幻莫测的网络负载。对网络访问的处理类似于文件访问，要尽可能多地缓存静态信息。
- 数学计算。如果程序中多处需要一个计算的结果，可以完成一次这个计算，并由多处共享其结果。
- 对象分配。如果需要在程序中创建和使用大量生命期很短的对象，可以考虑使用一个对象池，这在 17.3.2 节介绍。
- 线程创建。这个任务也很慢。可以把线程“缓存”到一个线程池中，请参见以下 17.3.3 节的介绍。

#### 缓存失效

缓存的一个常见问题是，保存的数据通常只是底层信息的副本。原数据可能在缓存过程中有所改变。例如，你可能想缓存一个配置文件中的值，这样就不必重复地读取这个配置文件了。不过，也许允许用户在程序运行期间修改配置文件，这就会使缓存的信息过时。在诸如此类的情况下，就需要一种缓存失效（cache invalidation）的机制，当底层数据修改时，要么必须停止使用缓存的信息，要么必须重新生成缓存。

缓存失效的一种技术是请求管理底层数据的实体在数据有所变化的时候通知程序。为此，可以通过一个回调（callback）做到，程序向管理器（即管理底层数据的实体）注册这个回调。还有另外一个技

术，程序可以轮询可能触发数据改变的某些事件，从而自动地重新生成缓存。不论你采用哪一种特定的缓存失效实现，都要保证一点：程序依赖一个缓存之前，先要充分考虑这些问题。

### 17.3.2 使用对象池

在第 13 章已经了解到，对象池技术可以避免在程序的生命期中创建和删除大量对象。如果知道程序需要同一类型的大量对象，而且对象的生命期都很短，就可以为这些对象创建一个池（pool）进行缓存。只要代码中需要一个对象，就可以向对象池请求。用完此对象时，要把它放回到池中。对象池只创建一次对象，因此它们的构造函数只调用一次，而不是每次使用时都调用。因此，当构造函数要完成一些设置动作，而且这些设置可以应用于该对象的多次使用时，对象池就很合适。另外在非构造函数方法调用中要在对象上设置特定于实例的参数时，也很适于采用对象池。

#### 一个对象池实现

本节提供了一个池类模板的实现，你可以在自己的程序中使用这个池实现。这个池在构造时会分配一大块（chunk）指定类的对象（译者注：这里的“块”可以理解为包括许多对象，即一堆对象），并通过 `acquireObject()` 方法交出对象。当客户用完这个对象时，会通过 `releaseObject()` 方法将其返回。如果调用了 `acquireObject()`，但是没有空闲的对象，池会分配另外一块对象。

对象池实现中最难的一方面是要记录哪些对象是空闲的，哪些对象正在使用。这个实现采用了以下做法，即把空闲的对象保存在一个队列中。每次客户请求一个对象时，池就会把队列中的第一个对象交给该客户。这个池不会显式地跟踪正在使用的对象。它相信客户在用完对象后会正确地将对象交还到池中。另外，这个池会在一个向量中记录所有已分配的对象。这个向量仅在撤销池时才会用到，以便释放所有对象的内存，从而避免内存泄漏。

这个代码使用了 `queue` 和 `vector` 的 STL 实现，这在第 4 章已经做过介绍。`queue` 容器允许客户用 `push()` 增加元素，用 `pop()` 删除元素，并用 `front()` 查看队头元素。`vector` 允许客户用 `push_back()` 增加元素。有关这两个容器的详细内容请参考第 21 章～第 23 章。

以下是类定义，并提供了注释来解释有关的细节。要注意，这个模板是基于相应的类类型（要在池中构造何种类型的对象）参数化的。

```
#include <queue>
#include <vector>
#include <stdexcept>
#include <memory>

using std::queue;
using std::vector;

//
// template class ObjectPool
//
// Provides an object pool that can be used with any class that provides a
// default constructor
//
// The object pool constructor creates a pool of objects, which it hands out
// to clients when requested via the acquireObject() method. When a client is
// finished with the object it calls releaseObject() to put the object back
// into the object pool.
//
// The constructor and destructor on each object in the pool will be called only
```

```

// once each for the lifetime of the program, not once per acquisition and release.
//
// The primary use of an object pool is to avoid creating and deleting objects
// repeatedly. The object pool is most suited to applications that use large
// numbers of objects for short periods of time.
//
// For efficiency, the object pool doesn't perform sanity checks.
// It expects the user to release every acquired object exactly once.
// It expects the user to avoid using any objects that he or she has released.
//
// It expects the user not to delete the object pool until every object
// that was acquired has been released. Deleting the object pool invalidates
// any objects that the user has acquired, even if they have not yet been released.
//
template <typename T>
class ObjectPool
{
public:
    //
    // Creates an object pool with chunkSize objects.
    // Whenever the object pool runs out of objects, chunkSize
    // more objects will be added to the pool. The pool only grows:
    // objects are never removed from the pool (freed), until
    // the pool is destroyed.
    //
    // Throws invalid_argument if chunkSize is <= 0
    //
    ObjectPool(int chunkSize = kDefaultChunkSize)
        throw(std::invalid_argument, std::bad_alloc);
    //
    // Frees all the allocated objects. Invalidates any objects that have
    // been acquired for use
    //
    ~ObjectPool();

    //
    // Reserve an object for use. The reference to the object is invalidated
    // if the object pool itself is freed.
    //
    // Clients must not free the object!
    //
    T& acquireObject();

    //
    // Return the object to the pool. Clients must not use the object after
    // it has been returned to the pool.
    //
    void releaseObject(T& obj);

protected:
    //
    // mFreeList stores the objects that are not currently in use
    // by clients.
    //
    queue<T*> mFreeList;

```

```

//
// mAllObjects stores pointers to all the objects, in use
// or not. This vector is needed in order to ensure that all
// objects are freed properly in the destructor.
//
vector<T*> mAllObjects;

int mChunkSize;
static const int kDefaultChunkSize = 10;

//
// Allocates mChunkSize new objects and adds them
// to the mFreeList
//
void allocateChunk();
static void arrayDeleteObject(T* obj);
private:
// Prevent assignment and pass-by-value.
ObjectPool(const ObjectPool<T>& src);
ObjectPool<T>& operator=(const ObjectPool<T>& rhs);
};

```

对于这个类定义有几点需要强调。首先，要注意，对象是按引用获取和释放的，而不是按指针，这样可以避免客户通过指针管理或释放对象。接下来，注意对象池的用户通过模板参数来指定可以创建哪一个类的对象（即指定类名），通过构造函数指定分配的“块大小”。这个“块大小”控制着一次可创建的对象数。以下是定义 `kDefaultChunkSize` 的代码：

```

template<typename T>
const int ObjectPool<T>::kDefaultChunkSize;

```

根据类定义，默认值 10 对于大多数使用来说可能都太小了。如果程序一次需要成千上万的对象，就应该使用一个更大、更合适的值。

构造函数验证 `chunkSize` 参数，并调用 `allocateChunk()` 辅助方法来得到起始的对象分配。

```

template <typename T>
ObjectPool<T>::ObjectPool(int chunkSize) throw(std::invalid_argument,
std::bad_alloc) : mChunkSize(chunkSize)
{
    if (mChunkSize <= 0) {
        throw std::invalid_argument("chunk size must be positive");
    }
    // Create mChunkSize objects to start.
    allocateChunk();
}

```

`allocateChunk()` 方法在连续的存储空间中分配 `mChunkSize` 个元素。它会在一个 `mAllObjects` vector 中存储对象数组的指针，并把各个对象压至 `mFreeList` queue。

```

//
// Allocates an array of mChunkSize objects because that's
// more efficient than allocating each of them individually.
// Stores a pointer to the first element of the array in the mAllObjects
// vector. Adds a pointer to each new object to the mFreeList.
//

```



```
template <typename T>
void ObjectPool<T>::allocateChunk()
{
    T* newObjects = new T[mChunkSize];
    mAllObjects.push_back(newObjects);
    for (int i = 0; i < mChunkSize; i++) {
        mFreeList.push(&newObjects[i]);
    }
}
```

析构函数只是释放 allocateChunk() 中分配的所有对象数组。不过，它使用了 for\_each() STL 算法来做到这一点，在此向 for\_each() 传递了一个 arrayDelete() 静态方法的指针，这个方法会对各个对象数组具体完成删除调用。如果不明白这段代码，有关详细内容请参考有关 STL 的第 21 章~第 23 章。

```
//
// Freeing function for use in the for_each algorithm in the
// destructor
//
template<typename T>
void ObjectPool<T>::arrayDeleteObject(T* obj)
{
    delete [] obj;
}

template <typename T>
ObjectPool<T>::~~ObjectPool()
{
    // Free each of the allocation chunks.
    for_each(mAllObjects.begin(), mAllObjects.end(), arrayDeleteObject);
}
```

acquireObject() 会返回空闲列表中的队头对象，如果没有空闲对象则首先调用 allocateChunk()。

```
template <typename T>
T& ObjectPool<T>::acquireObject()
{
    if (mFreeList.empty()) {
        allocateChunk();
    }
    T* obj = mFreeList.front();
    mFreeList.pop();
    return (*obj);
}
```

最后，releaseObject() 将对象返回到空闲列表的队尾。

```
template <typename T>
void ObjectPool<T>::releaseObject(T& obj)
{
    mFreeList.push(&obj);
}
```

### 使用对象池

请考虑一个要从用户得到请求并处理这些请求的应用。这个应用很可能是图形前端和后台数据库之间的一个中间件。例如，这可能是一个航空预订系统或一个在线银行应用的一部分。你可能想把每个用



户请求编码到一个对象中，这个类可能如下所示。

```
class UserRequest
{
public:
    UserRequest() {}
    ~UserRequest() {}

    // Methods to populate the request with specific information
    // Methods to retrieve the request data
    // (not shown)

protected:
    // Data members (not shown)
};
```

这里不用在程序的整个生命期中创建和删除大量请求，而是可以使用一个对象池。程序结构可能如下所示：

```
UserRequest& obtainUserRequest(ObjectPool<UserRequest>& pool)
{
    // Obtain a UserRequest object from the pool.
    UserRequest& request = pool.acquireObject();

    // Populate the request with user input
    // (not shown).

    return (request);
}

void processUserRequest(ObjectPool<UserRequest>& pool, UserRequest& req)
{
    // Process the request
    // (not shown).

    // Return the request to the pool.
    pool.releaseObject(req);
}

int main(int argc, char** argv)
{
    ObjectPool<UserRequest> requestPool(1000);

    // Set up program
    // (not shown).

    while (/* program is running */) {
        UserRequest& req = obtainUserRequest(requestPool);
        processUserRequest(requestPool, req);
    }

    return (0);
}
```

### 17.3.3 使用线程池

线程池与对象池很相似，即不是在程序的整个生命期中动态地创建和删除线程，而是创建一个线程池，按需使用池中的线程。如果程序要处理到来的网络请求，这种程序中常常会使用这个技术。Web 服务器就可以维护一个线程池，以备查找页面，从而对到来的各个客户请求做出响应。

线程支持是特定于平台的，这将在第 18 章将更深入地讨论，所以这里不提供线程池的例子。不过，完全可以照着对象池的方式编写一个线程池。

## 17.4 测评分析

尽管我们强烈建议你在设计和编写代码时就考虑效率，不过应该承认，并非每一个完成的应用都能有如期的表现。在强调程序的功能时很容易把效率摆在一边，以至于效率大幅下降。从我们的经验看，大多数效率优化都是针对已经在运作的程序。即使在开发中确实考虑了效率，但优化可能并非针对程序中最需要优化的部分。在第 4 章中曾经提到，大多数程序 90% 的时间都只是花在运行 10% 的代码上。这说明你可能优化了 90% 的代码，但是程序的运行时间只得到了 10% 的改善。很显然，对于程序所要应对的特定工作负载，程序中哪些代码执行得最多，你肯定就希望优化这部分代码。

因此，如果能对程序进行测评分析 (profile)，确定代码中的哪些部分需要优化，这往往很有帮助。已经有许多可用的测评工具 (profiling tool)，可以在程序运行时对程序进行分析，从而生成其性能的相关数据。大多数测评工具会指出程序中各函数所花的时间 (或占总执行时间的百分比)，以便在函数级提供分析。针对一个程序运行测评工具后，通常可以很快得出程序的哪些部分需要优化。优化前和优化后都做测评分析，这也很有用，可以证明优化是否确实有效果。

从我们的经验看，最好的测评工具是 IBM 的 Rational Quantify。它的许可费用很高，不过你应该查看一下你的公司或学术机构是否有这个工具的使用许可。如果明令禁止使用，还有许多免费的测评工具可供使用。其中最著名的就是 gprof，大多数 UNIX 系统上都可以找到这个工具，包括 Solaris 和 Linux。

### 利用 gprof 的测评例子

通过一个实际的编码例子最能体现测评分析的强大作用。提前声明，在此所示的第一个尝试中，存在的性能 bug 并不严重。真正的效率问题可能更为复杂，不过，要反映这么复杂的效率问题，往往需要很长的程序，而太长的程序是本书篇幅不能允许的。

假设你为美国社会保障部门工作。每年该部门都会提供一个 Web 网站，允许用户查看上一年以来新生儿的姓名。你的工作就是编写一个后台程序，为用户查找姓名。你的输入是一个文件，其中包含每个新生儿的姓名。显然这个文件可能包含重复的姓名。例如，在 2003 年的男孩文件中，Jacob 最普遍，出现了 29 195 次。程序必须读取这个文件来构造一个内存中数据库。然后用户就可以根据一个给定名 (或者该名字在所有婴儿中的归类) 来请求同名婴儿的绝对人数。

#### 第一次设计尝试

这个程序的逻辑设计包括一个 NameDB 类，这个类有以下公共方法：

```
#include <string>
#include <stdexcept>

using std::string;

class NameDB
```

```

{
    public:
        // Reads the list of baby names in nameFile to populate the database.
        // Throws invalid_argument if nameFile cannot be opened or read.
        NameDB(const string& nameFile) throw (std::invalid_argument);

        // Returns the rank of the name (1st, 2nd, etc).
        // Returns -1 if the name is not found.
        int getNameRank(const string& name) const;

        // Returns the number of babies with this name.
        // Returns -1 if the name is not found.
        int getAbsoluteNumber(const string& name) const;

        // Protected and private members and methods not shown
};

```

难点在于要为内存中数据库选择一个合适的数据结构。第一个尝试可能会选择姓名/人数对的一个数组，或者是 STL 中的向量。向量中的每一项会存储一个姓名，以及该姓名出现在原始数据文件中的次数。以下是有此设计的完整类定义：

```

#include <string>
#include <stdexcept>
#include <vector>

using std::string;

class NameDB
{
    public:
        NameDB(const string& nameFile) throw (std::invalid_argument);

        int getNameRank(const string& name) const;
        int getAbsoluteNumber(const string& name) const;

    protected:
        std::vector<std::pair<string, int> > mNames;

        // Helper methods
        bool nameExists(const string& name) const;
        void incrementNameCount(const string& name);
        void addNewName(const string& name);

    private:
        // Prevent assignment and pass-by-value.
        NameDB(const NameDB& src);
        NameDB& operator=(const NameDB& rhs);
};

```

需要注意这里使用了 STL vector 和 pair。pair 只是一个结合两个不同类型变量的工具类。有关 STL 的详细内容请参见第 21 章~第 23 章。

以下是构造函数和辅助函数 nameExists()、incrementNameCount() 和 addNewName() 的实现。如果你对 STL 不熟悉，可能会不明白 nameExists() 和 incrementNameCount() 中的循环。这些循环只是迭代处理 mNames 向量的所有元素。

```

//
// Reads the names from the file and populates the database.
// The database is vector of name/count pairs, storing the
// number of times each name shows up in the raw data,
//
NameDB::NameDB(const string& nameFile) throw (invalid_argument)
{
    // Open the file and check for errors.
    ifstream inFile(nameFile.c_str());
    if (!inFile) {
        throw invalid_argument("Unable to open file\n");
    }

    // Read the names one at a time.
    string name;
    while (inFile >> name) {
        // Look up the name in the database so far.
        if (nameExists(name)) {
            // If the name exists in the database, just
            // increment the count.
            incrementNameCount(name);
        } else {
            // If the name doesn't yet exist, add it with
            // a count of 1.
            addNewName(name);
        }
    }
    inFile.close();
}

//
// nameExists
//
// Returns true if the name exists in the database. Returns false otherwise.
//
bool NameDB::nameExists(const string& name) const
{
    // Iterate through the vector of names looking for the name.
    for (vector<pair<string, int> >::const_iterator it = mNameNames.begin();
         it != mNameNames.end(); ++it) {
        if (it->first == name) {
            return (true);
        }
    }
    return (false);
}

//
// incrementNameCount
//
// Precondition: name exists in the vector of names.
// Postcondition: the count associated with name is incremented.
//
void NameDB::incrementNameCount(const string& name)
{
    for (vector<pair<string, int> >::iterator it = mNameNames.begin();
         it != mNameNames.end(); ++it) {
        if (it->first == name) {

```

```

        it->second++;
        return;
    }
}

//
// addNewName
//
// Adds a new name to the database
//
void NameDB::addNewName(const string& name)
{
    mNames.push_back(make_pair<string, int>(name, 1));
}

```

在上例中需要注意，对于 `nameExists()` 和 `incrementNameCount()` 中的循环，也可以使用一个类似 `find_if` 的算法来完成同样的工作。在此我们专门选用了循环，目的就是为了强调性能问题。

聪明的读者可能已经注意到了一些性能问题。如果有数百万个姓名该怎么办？填充数据库时涉及的许多线性搜索会变得很慢。

作为一个完整的例子，以下给出了两个公共方法的实现：

```

//
// getNameRank
//
// Returns the rank of the name.
// First looks up the name to obtain the number of babies with that name.
// Then iterates through all the names, counting all the names with a higher
// count than the specified name. Returns that count as the rank.
//
int NameDB::getNameRank(const string& name) const
{
    // Make use of the getAbsoluteNumber() method.
    int num = getAbsoluteNumber(name);
    // Check if we found the name.
    if (num == -1) {
        return (-1);
    }
    //
    // Now count all the names in the vector that have a
    // count higher than this one. If no name has a higher count,
    // this name is rank number 1. Every name with a higher count
    // decreases the rank of this name by 1.
    //
    int rank = 1;
    for (vector<pair<string, int> >::const_iterator it = mNames.begin();
         it != mNames.end(); ++it) {
        if (it->second > num) {
            rank++;
        }
    }
    return (rank);
}
//

```



```
// getAbsoluteNumber
//
// Returns the count associated with this name
//
int NameDB::getAbsoluteNumber(const string& name) const
{
    for (vector<pair<string, int> >::const_iterator it = mNameMap.begin();
         it != mNameMap.end(); ++it) {
        if (it->first == name) {
            return(it->second);
        }
    }
    return (-1);
}
```

### 第一次尝试的测评分析

为了测试这个程序，需要一个主（main）函数。

```
#include "NameDB.h"

int main(int argc, char** argv)
{
    NameDB boys("boys_long.txt");

    cout << boys.getNameRank("Daniel") << endl;
    cout << boys.getNameRank("Jacob") << endl;
    cout << boys.getNameRank("William") << endl;

    return (0);
}
```

这个 main 函数创建了一个名为 boys 的 NameDB 数据库，告诉它用文件 boys\_long.txt 来填充数据。这个文件包含 500 500 个姓名。

使用 gprof 有 3 个步骤：

1. 编译程序时带一个特殊的标志，从而在下一次运行时可以记录原执行信息。在 Solaris 9 上，若使用 SunOne Studio 8 C++ 编译器，所带的标志是 -xpg：

```
> CC -o namedb -xpg main.cpp NameDB.cpp
```

2. 接下来，运行程序。此次运行应当在工作目录中生成一个文件 gmon.out。

3. 最后一步是运行 gprof 命令，来分析 gmon.out 测评信息，并生成可读的报告（有一定可读性）。-C 选项告诉 gprof 要重排（demangle）C++ 函数名，使之更可读。gprof 会输出至标准输出，所以应当把输出重定向到一个文件：

```
> gprof -C namedb gmon.out > gprof_analysis.out
```

下面可以分析数据了。遗憾的是，输出文件还有些不好理解。要了解如何解释这个输出要稍稍花点时间。gprof 提供了两组不同的信息。第二组信息总结了执行程序中各函数所花的时间。第一组信息也是更有用的一组信息，总结了执行各函数及其子孙函数（and its descendent）（译者注：这里的子孙与继承没有任何关系，只是表示函数中会调用其他函数，而所调用函数本身也可能调用其他函数，这些都称为顶层函数的子孙函数）所花费的时间。以下是 gprof\_analysis.out 文件中的部分输出，这里做了一些编



辑，使之更具可读性：

```
[2]      85.1    0.00    48.21    1    main [2]
```

上一行显示了 `main()` 及其子孙函数占用了程序总执行时间的 85.1%，共 48.21 秒。余下的 14.9% 则是完成其他任务花费的时间，如查找动态链接库和初始化全局变量。下一项显示了 `NameDB` 构造函数及其子孙函数花了 48.18 秒，这几乎就是 `main()` 的全部时间。`NameDB::NameDB` 之下的嵌套项显示出哪一个子孙函数花的时间最多。在此可以看到 `nameExists()` 和 `incrementNameCount()` 都花了大约 14 秒。要记住，这些时间都是所有函数调用的时间总和。这些行中的第三列显示了函数调用次数（`nameExists()` 的调用次数为 500 500，`incrementNameCount()` 的调用次数为 499 500）。其他函数不会占用 `NameDB` 的太多时间。

```
[3]      85.1    0.03    48.18    1    NameDB::NameDB
          9.60    14.04  500500/500500    bool NameDB::nameExists
          8.36    14.00  499500/499500    void NameDB::incrementNameC
ount
```

先不继续深入分析，在此你应该注意到两点：

1. 向数据库填充大约 500 000 个名字需要 48 秒，这太慢了。也许需要一个更好的数据结构。
2. `nameExists()` 和 `incrementNameCount()` 所占的时间几乎相同，而且调用的次数也几乎一样。如果考虑一下这个程序的应用领域，应该不难理解，文本文件输入中的大多数名字都是重复的，因此，对 `nameExists()` 的大量调用都相应地会带来了一个对 `incrementNameCount()` 的调用。如果回过头去看代码，可以看到这些函数基本上相同。也许它们可以合并。另外，它们所做的工作大多都是搜索 `vector`。如果使用一个有序数据结构来减少搜索时间可能更好一些。

### 第二次尝试

基于对 `gprof` 输出的两点观察，下面可以重新设计这个程序了。新设计使用一个 `map` 而不是 `vector`。第 4 章曾经指出，STL `map` 采用了一个底层树结构以保证数据项有序，而且可以提供  $O(\log n)$  的查找，而不是向量中的  $O(n)$  查找。

新版本的程序还把 `nameExists()` 和 `incrementNameCount()` 合并为一个 `nameExistsAndIncrement()` 函数。

以下是这个新的类定义：

```
#include <string>
#include <stdexcept>
#include <map>

using std::string;

class NameDB
{
public:
    NameDB(const string& nameFile) throw (std::invalid_argument);

    int getNameRank(const string& name) const;
    int getAbsoluteNumber(const string& name) const;

protected:
    std::map<string, int> mNames;
```

```
bool nameExistsAndIncrement(const string& name);
void addNewName(const string& name);
```

```
private:
    // Prevent assignment and pass-by-value
    NameDB(const NameDB& src);
    NameDB& operator=(const NameDB& rhs);
};
```

下面给出了新的方法实现：

```
//
// Reads the names from the file and populates the database.
// The database is a map associating names with their frequency.
//
NameDB::NameDB(const string& nameFile) throw (invalid_argument)
{
    //
    // Open the file and check for errors.
    //
    ifstream inFile(nameFile.c_str());
    if (!inFile) {
        throw invalid_argument("Unable to open file\n");
    }
    //
    // Read the names one at a time.
    //
    string name;
    while (inFile >> name) {
        //
        // Look up the name in the database so far.
        //
        if (!nameExistsAndIncrement(name)) {
            //
            // If the name exists in the database, the
            // function incremented it, so we just continue.
            // We get here if it didn't exist, in case which
            // we add it with a count of 1.
            //
            addNewName(name);
        }
    }
    inFile.close();
}
```

```
//
// nameExistsAndIncrement
//
// Returns true if the name exists in the database. false
// otherwise. If it finds it, it increments it.
//
bool NameDB::nameExistsAndIncrement(const string& name)
{
    //
    // Find the name in the map.
    //
```

```

    map<string, int>::iterator res = mNameMap.find(name);
    if (res != mNameMap.end()) {
        res->second++;
        return (true);
    }
    return (false);
}

```

```

//
// addNewName
//
// Adds a new name to the database
//
void NameDB::addNewName(const string& name)
{
    mNameMap.insert(make_pair<string, int>(name, 1));
}

```

```

//
// getNameRank
//
// Returns the
int NameDB::getNameRank(const string& name) const
{
    int num = getAbsoluteNumber(name);

    //
    // Check if we found the name.
    //
    if (num == -1) {
        return (-1);
    }

    //
    // Now count all the names in the map that have
    // count higher than this one. If no name has a higher count,
    // this name is rank number 1. Every name with a higher count
    // decreases the rank of this name by 1.
    //
    int rank = 1;
    for (map<string, int>::const_iterator it = mNameMap.begin();
         it != mNameMap.end(); ++it) {
        if (it->second > num) {
            rank++;
        }
    }
}

```

```

    return (rank);
}

//
// getAbsoluteNumber
//
// Returns the count associated with this name
//
int NameDB::getAbsoluteNumber(const string& name) const

```

```
{
    map<string, int>::const_iterator res = mNameMap.find(name);
    if (res != mNameMap.end()) {
        return (res->second);
    }

    return (-1);
}
```

### 第二次尝试的测评分析

还是按前面所示的步骤，可以针对新版本的程序得到 gprof 性能数据。这个数据很让人高兴。

[2]	85.3	0.00	3.19	1	main [2]
-----	------	------	------	---	----------

main() 现在只花了 3.19 秒，得到了 15 倍的提升！对这个程序肯定还能进一步改进，不过我们把进一步的改进作为练习留给读者来完成。可以提示一点，缓存可能有助于姓名归类。

## 17.5 小结

本章讨论了 C++ 程序中效率和性能的主要方面，而且还提供了设计和编写更高效应用的一些具体提示和技术。希望你能对性能的重要性有所认识，并认识到测评工具的强大功能。要记住，应当从程序生命期的一开始就考虑性能和效率，设计级效率远比语言级效率更重要。

## 第 18 章 开发跨平台和跨语言的应用

C++ 程序可以编译为能在多种不同计算平台上运行，而且 C++ 语言得到了严格的定义，可以确保在一个平台上用 C++ 编程与在另一个平台上用 C++ 编程没有太大出入。不过，尽管对这种语言做了标准化，但是在用 C++ 编写专业质量的程序时，平台间的差异最终还是会反映出来。即使开发仅限于一个特定平台，编译器的一些小差异也会导致很严重的编程问题。这一章将介绍在多平台和多编程语言环境中编程时存在的复杂性。

本章第一部分会分析 C++ 程序员所遇到的与平台相关的问题。平台 (platform) 是指构成开发系统和/或运行时系统的所有部分的集合。例如，你的平台可能是一个运行在 Windows XP (采用 Pentium 处理器) 上的 Microsoft C++ 编译器。或者，你的平台也可能是运行在 Linux (采用 PowerPC 处理器) 上的 gcc 编译器。这两个平台都能编译和运行 C++ 程序，不过二者之间存在着一些显著的差异。

本章第二部分介绍了 C++ 如何与其他编程语言交互。尽管 C++ 是一种通用语言，但是可能并非完成特定任务的最佳选择。可以通过多种机制将 C++ 与其他可能更能满足需求的语言相集成。

### 18.1 跨平台开发

C++ 语言为什么会遇到平台问题，这有许多原因。尽管 C++ 是一种高级语言，但是其定义包含有低级实现细节。例如，C++ 数组定义为保存在连续的内存块中。这个特定的实现细节就使得 C++ 语言暴露出一个可能的问题：并非所有系统都采用同样的方式来组织和管理内存。C++ 还要面对另一个难题，尽管它提供了一个标准语言和一个标准库，但是没有提供标准实现。对于不同的 C++ 编译器和库开发商，会对规范有不同的解释，这样在从一个系统转向另一个系统时就可能带来麻烦。最后一点，C++ 作为标准提供的功能还不算完备，因此 C++ 语言只是选择性的。尽管 C++ 提供了一个标准库，但是复杂的程序通常还需要 C++ 语言未能提供的功能。这些功能一般由第三方库或平台提供，这就可能存在很大差别。

#### 18.1.1 体系结构问题

体系结构 (architecture) 一词通常指程序运行所在的处理器或处理器组。运行 Windows 或 Linux 的标准 PC 通常在 x86 体系结构上运行，而 Mac OS 通常在 PowerPC 体系结构上运行。作为一种高级语言，C++ 屏蔽了这些体系结构之间的差异。例如，Pentium 处理器完成某个功能可能有一个指令，而 PowerPC 完成同一功能则有 6 个指令。作为一个 C++ 程序员，不必了解这些差别究竟是什么，甚至无需知道存在这样一些差别。使用高级语言的一个好处就是编译器会负责将代码转换为处理器的本地汇编码格式。

不过，有时在 C++ 代码中确实会反映出处理器差别。除非你在做相当低级的工作，否则可能并不会遇到大多数这样的问题，不过要知道这些问题的客观存在。

##### 二进制兼容性

你也许已经知道，如果为一个 Pentium 计算机编写并编译了一个程序，那么这个程序不能在一个 Mac 上运行。这两个平台不是二进制兼容的 (binary compatible)，因为它们的处理器不支持相同的指令

集。应该记得，在编译一个 C++ 程序时，源代码会转换为二进制指令由计算机执行。这种二进制格式由平台定义，而不是由 C++ 语言定义。

二进制兼容性问题的解决方案通常是跨编译 (cross-compiling)。在跨编译一个程序时，会对各个可能的目标体系结构分别构建一个单独的版本。有些编译器直接支持跨编译。另外一些编译器则需要在目标体系结构上分别构建各个版本。

对于二进制表示中存在的差别，还有一种解决方案，这就是开源发布 (open source distribution)。让最终用户能够得到源代码，他就能在他的系统上完成本地编译，并构建一个程序版本，这个程序会采用适于他机器的二进制格式。如第 4 章所述，最近几年，开源软件越来越流行。一个主要原因就是程序员可以协同开发软件，并使得软件能在更多的平台上运行。

### 字和类型大小

字 (word) 是计算机体系结构的基本存储单元。在大多数系统中，字就是一个地址和/或一个处理器指令的大小。如果有人把一个体系结构描述为 32 位，很可能是指字大小为 32 位或 4 字节。一般地，如果系统的字大小较大，就能处理更多内存，并能更快地处理复杂程序。

由于指针是内存地址，因此指针的大小自然就是字大小。许多程序员了解到的都是指针总是 4 字节，但事实上并不一定如此。例如，考虑以下程序，它会输出一个指针的大小。

```
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    int *ptr;

    cout << "ptr size is " << sizeof(ptr) << " bytes" << endl;
}
```

如果这个程序在一个 32 位的 Pentium 体系结构上运行，输出是：

```
ptr size is 4 bytes
```

在一个 64 位的 Itanium 系统上运行时，输出则是：

```
ptr size is 8 bytes
```

从程序员的观点看，指针大小不同的后果只是不能将指针与 4 字节等同。更一般地，注意，C++ 标准并没有预先确定大多数类型的大小。标准中只是指出短整型所占空间与整型相同（或更少），而整型所占空间与长整型相同（或更少）。整型本身则认为包含了足以保存一个字的空间，在前面已经看到，这个大小可能会有变化。

### 字序

所有现代计算机都以二进制表示来存储数字，不过同一个数在两个平台上的表示可能并不一样。这听上去有些矛盾，不过可以看到，有两种读取数字的方法，而且这两种方法都是有道理的。

计算机内存中的一个槽通常是一个字节，因为大多数计算机都是按字节寻址的。C++ 中的数字类型往往是多字节。例如，一个 short 可能是 2 字节。假设程序包含以下这行代码：

```
short myShort = 513;
```



采用二进制表示，数字 513 就表示为 0000001000000001。这个数包含 16 个 1/0，即包含 16 位。由于一个字节有 8 位，计算机就需要两个字节来存储这个数。因为每个内存地址包含 1 字节，计算机需要把这个数分成多字节。假设一个 short 是 2 字节，这个数就可以分成平均的两部分。数字的较高部分置于高位字节 (high-order byte)，数字的较低部分置于低位字节 (low-order byte)。在这个例子中，高位字节是 00000010，而低位字节是 00000001。

既然这个数已经分成了满足内存大小的部分，现在惟一的问题就是如何将其存储在内存中。需要两个字节，但是字节的顺序不确定，而且实际上字节的顺序要取决于当前系统的体系结构。

表示这个数的一种做法是，先在内存中放置高位字节，然后再放低位字节。这种策略称为 *big-endian* 序 (*big-endian ordering*)，因为先放的是数字中较大的部分。PowerPC 和 Sparc 处理器就使用一种 *big-endian* 方法。有些处理器 (如 x86) 则以相反的顺序放置字节，即先在内存中放入低位字节。这种方法称为 *little-endian* 序 (*little-endian ordering*)，因为先放的是数字中较小的部分。体系结构可以在二者中任意选择，通常会基于向后兼容性做出选择。有意思的是，“*big-endian*”和“*little-endian*”这两个词早在现代计算机诞生之前的数百年就已经出现了。Jonathan Swift 在他的 18 世纪小说《格列佛游记》中率先用这两个词来描述争论一个可笑问题的两大阵营，他们争论的是应该从鸡蛋的哪一头剥才合适。许多计算机科学家认为当前对于高位字节低位字节哪个在前的讨论与 Swift 小说中描述的这个问题至少同样愚蠢。

不论一个特定体系结构使用何种字序 (word ordering)，程序都可以继续正常地使用数字值，而不用担心机器到底使用 *big-endian* 序还是 *little-endian* 序。只有当数据要跨体系结构迁移时才会反映出字序问题。例如，如果你在跨网络发送二进制数据，可能就需要考虑另一个系统的字序。类似地，如果你在向一个文件写二进制数据，可能要考虑如果在另一个系统上打开这个文件，而该系统有完全相反的字序，会出现什么情况。

### 18.1.2 实现问题

编写 C++ 编译器时，会由想遵循 C++ 标准的人来设计。遗憾的是，C++ 标准有数百页之多，包括有平淡的描述、语言文法和例子。即使两个人都按照这个标准来实现一个编译器，也往往不会恰好以同样的方式解释标准中指定的每一条信息，或者不太可能同样地把握第一种边缘情况。尽管很难相信，但是必须承认，即使是编译器本身也存在 bug。

#### 编译器疑难问题和扩展

你遇到的第一个编译器 bug 可能是超现实的体验。你可能多年以来一直在跟踪和修正自己的 bug，但最后才发现你所依赖的这个程序本身存在缺陷！C++ 编译器自 C++ 语言诞生以来已经得到了很大改进，但是 C++ 编译器中 bug 依然存在。最起码是对规范有不同的解释，或者编译器中遗漏了某些语言特性。不过，你可能会时不时地发现，编译器甚至会做错事。

要找出或避免编译器 bug 并没有简单的规则。能做的无非是保持编译器及时更新，还可能需订购有关编译器的邮件列表或新闻组。如果怀疑自己遇到了一个编译器 bug，通过在网上搜索你看到的错误消息或情况，可能就会找到一种解决方法或补丁。

编译器在最新增加的语言特性方面往往存在问题，因此口碑不佳。例如，C++ 中有些模板和运行时类型特性原来并不是语言的一部分。第 11 章曾提到，有些编译器还不能适当地支持这些特性。

要注意的另一个问题是编译器通常包括自己的语言扩展，但没有明白地告知程序员。例如，基于栈的可变大小数组并不是 C++ 语言的一部分，但是以下两行代码在 g++ 编译器上可以编译。

```
int i = 4;
char myStackArray[i]; // Not a standard language feature!
```

有些编译器扩展可能很有用，但是如果你以后某个时候要换编译器，就应该查看编译器是否有一个严格的模式会禁止这种扩展。例如，用 g++ 编译上述代码时提供 pedantic 标志，就会得到以下警告：

```
warning: ISO C++ forbids variable-size array 'myStackArray'
```

C++ 规范通过 #pragma 机制支持一类编译器定义的语言扩展。#pragma 是一个预编译器指令，其行为由具体实现定义。如果实现不理解这个指令，就会将其忽略。例如，有些编译器允许程序员暂时用 #pragma 把编译器警告关掉。不过，这种行为取决于具体的编译器，不要完全依赖它。

### 库实现

你的编译器可能包括 C++ 标准库的一个实现，其中包括标准模板库，这种情况很常见。不过，由于 STL 是用 C++ 编写的，没有必要一定使用与编译器捆绑的 STL 库。如果有在速度方面更优化的第三方 STL，完全可以使用这样一个库，甚至还可以编写自己的 STL。

当然，与编写编译器的人一样，实现 STL 的人也会面对同样的问题，对标准的解释会因各人主观而定。另外，某些实现可能会做一些折衷，以至于不能满足需求。例如，有些实现可能会放弃多处理器支持来提升单处理器性能测评结果。另外一些实现则可能专门为多处理器做了优化。

在使用一个 STL 实现时，实际上在使用任何第三方库时，都要考虑设计者在开发时所做的折衷，这很重要。如果库是开源的，可能可以找到有关开发问题的一个列表或一个 bug 数据库。第 4 章对使用库所涉及的问题提供了更详细的介绍。

### 18.1.3 特定于平台的特性

C++ 是一种强大的通用语言。随着标准库的增加，C++ 语言已经配备了如此多的特性，如果程序员不是太钻研，完全有可能好几年没有任何麻烦地使用 C++ 编写代码，而从未超出过 C++ 提供的功能范围。不过，专业的程序可能会要求一些 C++ 并未提供的功能。这一节将列出平台所提供（而非 C++ 提供）的一些重要特性。

- 图形用户界面。如今，大多数商业程序都在拥有一个图形用户界面的操作系统上运行，其中包含诸如可点击的按钮、可移动的窗口和层次式菜单等元素。C++ 类似于 C 语言，并没有这些元素的概念。要用 C++ 写一个图形化应用，需要使用特定于平台的库，这样才能绘制窗口、通过鼠标接受用户输入，以及完成其他图形化任务。第 25 章介绍了面向对象图形化框架，这就是平台提供此功能的一种途径。
- 网络。我们编写应用的方式因 Internet 已经有所改变。如今，应用通过 Web 查看是否存在更新，或者游戏提供一个网络化多玩家模式，这些都不算少见。C++ 没有为网络提供机制，但是存在许多标准库。编写网络软件最常见的方法是通过一个称为 socket（套接字）的抽象。在大多数平台上都可以找到 socket 库实现，它提供了一种简单的面向过程的方法在网络上传送数据。有些平台支持一种基于流的网络系统，其操作类似于 C++ 中的 I/O 流。
- OS 事件和应用交互。在纯 C++ 代码中，很少与外部操作系统和其他应用交互。如果没有平台扩展，能提供给标准 C++ 程序的只是命令行参数而已。例如，复制和粘贴之类的操作在 C++ 中就没有直接提供支持，而需要平台提供的库。
- 低级文件。在第 14 章中，你了解了 C++ 中的标准 I/O，包括读写文件。许多操作系统都提供了它们自己的文件 API，有时这些 API 可能与 C++ 中的标准文件类不兼容。这些库通常提供了特定于操作系统的文件工具，如得到当前用户主目录的机制，或访问操作系统配置文件的机制等。一般说来，一旦开始使用一个特定平台的 API，就应该把 C++ I/O 类转换为该平台的 I/O 类（如果存在）。

- 线程。在 C++ 中并未直接对程序中并发线程的执行提供支持。其实现相当依赖于操作系统的内部工作，所以 C++ 语言中没有包括线程。最常用的线程库称为 pthreads。许多操作系统和面向对象框架也提供了它们自己的线程模型。

## 18.2 跨语言开发

对于某些类型的程序，C++ 可能并非完成任务的最佳工具。例如，如果 UNIX 程序需要与 shell 环境密切交互，那么最好是编写一个 shell 脚本而不是 C++ 程序。如果程序会完成大量文本解析，可能会认为 Perl 语言才是上选。有时你想要的是一种混合语言，希望它混合了 C++ 的一般特性以及另一种语言的特有特性。幸运的是，确实有一些技术可以让你尽享这两个领域的优势，既有 C++ 的灵活性，还有另一种语言的独有特色。

### 18.2.1 混合 C 和 C++

你已经知道了，C++ 语言是 C 语言的一个超集。除了少许例外，所有 C 程序都能在 C++ 中顺利编译和运行。这些例外往往与保留字有关。例如，在 C 中，class 这个词没有特殊的含义。因此，可以把它当作一个变量名，如以下 C 程序所示。

```
#include <stdio.h>

int main(int argc, char** argv)
{
    int class = 1; // Compiles in C, not C++

    printf("class is %d\n", class);
}
```

这个程序能在 C 中编译并运行，但是如果作为 C++ 代码编译则会得到一个错误。将一个程序从 C 转换或移植到 C++ 时，往往会遇到此类错误。幸运的是，这个问题的修正通常相当简单。在这个例子中，只需要把 class 变量重命名为 classID，以上代码就能编译了。

由于能很容易地在 C++ 程序中结合 C 代码，当遇到一个用 C 编写的有用的库或传统代码时，这就会很方便。函数和类可以很好地一同工作，这在本书中可谓屡见不鲜。类方法可以调用一个函数，函数也可以利用对象。

### 18.2.2 转换模式

将 C 和 C++ 混合时，存在的危险之一是程序可能开始丧失面向对象性。例如，如果面向对象 Web 浏览器利用一个过程性网络库来实现，程序就会混杂这两种模式。由于在这个应用中网络任务非常重要，而且所占的比重很大，你可能会考虑在过程性库之外编写一个面向对象包装器。

例如，假设你在用 C++ 编写一个 Web 浏览器，但是使用了一个 C 网络库，其中包含了以下代码声明的函数。需要注意，为简单起见，这里省略了 HostRecord 和 Connection 数据结构。

```
// netwrklib.h

#include "hostrecord.h"
#include "connection.h"

/**
 * Gets the host record for a particular Internet host given
```

```

    * its hostname (i.e. www.host.com)
    */
    HostRecord* lookupHostByName(char* inHostName);

    /**
     * Connects to the given host
     */
    Connection* connectToHost(HostRecord* inHost);

    /**
     * Retrieves a Web page from an already-opened connection
     */
    char* retrieveWebPage(Connection* inConnection, char* page);

```

netwrklib.h 接口相当简单和直接。不过，它不是面向对象的，而且使用这样一个库的 C++ 程序员往往会觉得很麻烦。这个库没有组织为一个内聚的类，甚至没有正确地加上 const！当然，一个好的 C 程序员可能会写出一个更好的接口，但是作为库的用户，给什么就必须接受什么。编写一个包装器就是定制此接口的一个机会。

在为这个库建立一个面向对象包装器之前，先来看一下如何加以使用，从而对实际使用有所了解。在以下程序中，使用了 netwrklib 库来获取 www.wrox.com/index.html 上的网页。

```

#include <iostream>
#include "netwrklib.h"

using namespace std;

int main(int argc, char** argv)
{
    HostRecord* myHostRecord = lookupHostByName("www.wrox.com");
    Connection* myConnection = connectToHost(myHostRecord);
    char* result = retrieveWebPage(myConnection, "/index.html");

    cout << "The result is " << result << endl;
}

```

为了让库更具有面向对象性，一种可能的办法是提供一个抽象，由它识别查找主机之间的链接（译者注：即查找适当的主机）、连接至该主机，并获取一个 Web 页面。一个好的面向对象包装器可以隐藏 HostRecord 和 Connection 类型不必要的复杂细节。

回忆第 3 章和第 5 章介绍过的设计原则，这个新类应当考虑到这个库的常见用例。前面的例子显示了最常使用的模式：首先查找一个主机，然后建立一个连接，接下来获取一个网页。还有可能会从同一个主机获取更多的网页，因此一个好的设计还应当提供这种使用模式。

以下是 WebHost 类定义的 public 部分。基于这个类，编写客户的程序员可以很容易地完成常见用例。

```

// WebHost.h

class WebHost {

public:
    /**
     * Constructs a WebHost object for the given host
     */

```

```

WebHost(const string& inHost);

/**
 * Obtains the given page from this host
 */
string getPage(const string& inPage);

};

```

请考虑客户程序员会以何种方式使用这个类。再重复前面用于 netwrklib 库的例子。

```

#include <iostream>
#include "WebHost.h"

int main(int argc, char** argv)
{
    WebHost myHost("www.wrox.com");
    string result = myHost.getPage("/index.html");

    cout << "The result is " << result << endl;
}

```

WebHost 类有效地封装了一个主机的行为，提供了有用的功能而没有不必要的调用和数据结构。这个类甚至提供了一个有用的新功能：一旦创建了一个 WebHost，就可以用它来获得多个 Web 页面，这样可以减少代码，而且可能会使程序运行得更快。

WebHost 类的实现大量使用了 netwrklib 库，但没有将其内部工作暴露给用户。为了支持这个抽象，该类需要一个数据成员，对头文件修改后，如下所示。

```

// WebHost.h

#include "netwrklib.h"

class WebHost {
public:
    /**
     * Constructs a WebHost object for the given host
     */
    WebHost(const string& inHost);

    /**
     * Obtains the given page from this host
     */
    string getPage(const string& inPage);

protected:
    Connection* mConnection;
};

```

相应的源文件为 netwrklib 库中包含的功能提供了一个新的接口。首先，构造函数为指定主机构建一个 HostRecord。因为 WebHost 类处理的是 C++ 字符串而不是 C 风格的字符串，它使用了 inHost 上的 c\_str() 方法来得到一个 const char\*，然后完成一个 const 强制转换，以此修补 netwrklib 在 const 方面的不足。所得到的 HostRecord 用于创建一个 Connection，它会存储在 mConnection 数据成员中以备以后使用。



```
WebHost::WebHost(const string& inHost)
{
    const char* host = inHost.c_str();

    HostRecord* theHost = lookupHostByName(const_cast<char*>(host));

    mConnection = connectToHost(theHost);
}
```

以后对 `getPage()` 的调用会把所存储的连接传递给 `netwrklib` 的 `retrieveWebPage()` 函数，并把结果值作为 C++ 字符串返回。

```
string getPage(const string& inPage)
{
    const char* page = inPage.c_str();

    string result = retrieveWebPage(mConnection, const_cast<char*>(page));

    return result;
}
```

了解网络的读者可能会注意到，与主机的连接若一直打开，这不是一个好的做法，而且未能遵循 HTTP 规范。不过，我们在这个例子中强调的是高雅性，而不是常规性（译者注：作者的意思是这种做法可以保证面向对象性，而没有强调是否满足最佳实践）。

可以看到，`WebHost` 类在 C 库外提供了一个面向对象包装器。通过提供一个抽象，可以修改底层实现，而不会影响客户代码，或者还可以提供额外的特性，如连接引用计数或页面的解析等。

### 18.2.3 与 C 代码的链接

在前面的例子中，假设已经有可用的原始 C 代码。这个例子充分利用了这样一个事实，即大多数 C 代码都能顺利地用 C++ 编译器编译。如果只有编译后的 C 代码，可能是以库的形式提供给你，此时仍然能在 C++ 程序中加以使用，不过为此需要多做几个步骤。

已编译的 C 代码与已编译 C++ 代码的格式不同，所以要告诉编译器某些函数是用 C 编写的，这样链接器就能适当地使用这些函数了。这是通过 `extern` 关键字做到的。

在以下代码中，`doCFunction()` 函数原型指定为一个外部 C 函数。

```
extern "C" {
    void doCFunction(int i);
}

int main(int argc, char** argv)
{
    // Call the C function.
    doCFunction(8);
}
```

`doCFunction()` 的具体定义在一个已编译库文件中提供，这个文件会在链接阶段联接。上述 `extern` 关键字只是通知编译器所链入的代码是用 C 编译的。

使用 `extern` 的一种更常见的模式是在头文件级使用。例如，如果你在使用一个用 C 编写的图形库，



它可能提供了一个. h 文件供你使用。你可以编写另一个头文件，将原来的头文件包在一个 extern 块中，从而指定整个头文件定义的函数都是用 C 编写的。作为包装器的. h 文件通常会命名为. hpp，从而与 C 版本的头文件相区别：

```
// graphicslib.hpp

extern "C" {
    #include "graphicslib.h"
}
```

不论是在 C++ 程序中包含 C 代码，还是与一个已编译的 C 库链接，都要记住，尽管 C++ 基本上是 C 的一个超集，但是不同的语言有不同的设计目标。可以调整 C 代码以便在 C++ 中正常工作，尽管这种做法也很常见，但是更好的做法是在过程性 C 代码外提供一个面向对象 C++ 包装器。

#### 18.2.4 利用 JNI 混合 Java 和 C++

尽管这是一本有关 C++ 的书，但我们并不会谎称这是目前最新最时髦的语言。20 世纪 90 年代中期，Java 语言为编程世界带来了一场风暴，而且从那以后，Java 已经得到迅速普及。Java 和 C++ 在某种程度上很相似，不过它们的能力不同。如果不搅和到信仰之争中，对于 C++，最常提到的一个优点就是它的速度，而 Java 最常提到的优点则是它为网络编程和图形界面提供的内置库。

Java 本地接口 (Java Native Interface, JNI) 是 Java 语言的一部分，利用 JNI，程序员可以访问不是用 Java 编写的功能。由于 Java 是一种跨平台语言，其设计初衷就是要让 Java 程序与操作系统交互。利用 JNI，程序员还可以利用采用其他语言编写的库，如 C++。如果 Java 程序员开发的应用对性能要求很高，或者需要传统代码，访问 C++ 库就会很有用。

JNI 还可以用于在一个 C++ 程序中执行 Java 代码，不过这种使用很少见。目前传统的 C++ 代码远比传统 Java 代码多，因此大多数使用 Java 代码的应用都是彻头彻尾的 Java 程序。因为这是一本有关 C++ 的书，我们并不打算介绍 Java 语言。本节所面向的是那些已经了解 Java，而且想在其 Java 代码中加入 C++ 代码的读者。

开始你的跨语言之旅时，先从 Java 程序开始。对于这个例子，只需提供最简单的 Java 程序就足够了：

```
public class HelloCpp {
    public static void main(String[] args)
    {
        System.out.println("Hello from Java!\n");
    }
}
```

下一步有点奇怪。需要声明一个将使用另一种语言编写的 Java 方法。为此，要使用 native 关键字，并省略其实现：

```
public class HelloCpp {
    // This will be implemented in C++.
    public native void callCpp();

    public static void main(String[] args)
    {
        System.out.println("Hello from Java!\n");
    }
}
```

C++代码最终会编译为一个共享库，它会动态加载到这个Java程序中。需要在一个Java静态模块内部加载这个库，这样当Java程序开始执行时就可以加载这个库。库名可以是任意的，这个例子中使用了hellocpp.so作为库名。用.so扩展名结尾的文件是UNIX系统上的共享库。Windows用户最常使用的是.dll文件。

```
public class HelloCpp {

    static {
        System.load("hellocpp.so");
    }

    // This will be implemented in C++.
    public native void callCpp();

    public static void main(String[] args)
    {
        System.out.println("Hello from Java!\n");
    }
}
```

最后，需要从Java程序中真正调用C++代码。callCpp() Java方法相当于尚未编写的C++代码的一个占位符。由于callCpp()是HelloCpp类的一个方法，Java程序只是创建一个新的HelloCpp对象，并调用callCpp()方法。

```
public class HelloCpp {

    static {
        System.load("hellocpp.so");
    }

    // This will be implemented in C++.
    public native void callCpp();

    public static void main(String[] args)
    {
        System.out.println("Hello from Java!\n");

        HelloCpp cppInterface = new HelloCpp();
        cppInterface.callCpp();
    }
}
```

在Java这边要做的就是这么多！现在，只需正常地编译这个Java程序：

```
javac HelloCpp.java
```

然后使用javah程序（本书作者喜欢把它念作jav-AHH!）为本地方法创建一个头文件。

```
javah HelloCpp
```

运行javah之后，会看到一个名为HelloCpp.h的文件，这是一个能完全正常工作的C/C++头文件（不过有些难看）。在这个头文件中，是一个Java\_HelloCpp\_callCpp()函数的C函数定义。C++程序需要实现这个函数，以便由Java程序调用。其完整签名是：

```
void Java_HelloCpp_callCpp (JNIEnv* env, jobject javaobj);
```

这个函数的 C++ 实现可以充分利用 C++ 语言的特性。这个例子只是从 C++ 输出一些文本。首先，需要包含 `jni.h` 头文件和 `javah` 创建的 `HelloCpp.h` 文件。如果还要使用其他 C 或 C++ 头文件，那么还需要包含这些头文件。

```
#include <jni.h>
#include "HelloCpp.h"
#include <iostream>
```

C++ 函数仍正常编写。要记住，你是在实现一个函数，而不是编写一个程序。在此不需要一个 `main()`。利用函数的参数，可以与 Java 环境以及调用本地代码的对象交互。这些内容未在这个例子中展示。

```
#include <jni.h>
#include "HelloCpp.h"
#include <iostream>
```

```
void Java_HelloCpp_callCpp(JNIEnv* env, jobject javaobj)
{
    std::cout << "Hello from C++!" << std::endl;
}
```

对此代码的编译取决于环境，不过往往需要调整计算机的设置，使之包含 JNI 头文件和本地 Java 库文件的位置。在 Linux 上使用 `gcc` 编译器，编译命令可能如下所示。

```
g++ -shared -I/usr/java/jdk/include/ -I/usr/java/jdk/include/linux HelloCpp.cpp \
-o hellocpp.so
```

编译器的输出是一个将由 Java 程序使用的库。只要这个共享库位于 Java 类路径中，就可以正常地执行 Java 程序了：

```
java HelloCpp
```

应该能看到以下结果：

```
Hello from Java!
```

```
Hello from C++!
```

当然，对于通过 JNI 能够做什么，这个例子只是触及了一点皮毛。可以使用 JNI 与操作系统特性或硬件驱动程序交互。要想全面了解 JNI，应当参考一本 Java 方面的参考书。

### 18.2.5 C++ 与 Perl 和 Shell 脚本的混合

C++ 包含一种内置的通用机制，可以与其他语言和环境交互，对此，你可能已经多次用到，但是未加注意。这就是 `main()` 函数的参数和返回值。

C 和 C++ 在设计时都考虑了命令行接口。`main()` 函数接收用户在命令行上键入的参数，并返回一个可由调用者解释的状态码。许多大型图形应用都会忽略 `main()` 的参数，因为图形界面往往要避免传递参数。不过，在一个脚本环境中，程序的参数可能是一种相当强大的机制，可以利用这种机制与环境交互。

#### 脚本 vs 编程

在深入讨论混合 C++ 和脚本的细节之前，先考虑一下你的项目究竟是一个应用还是一个脚本。其区别很微妙，而且这方面还存在争议。以下描述只是一些指导原则。许多所谓的脚本与完整的应用同样复

杂。问题并不是某个工作能不能作为一个脚本来完成，而是脚本语言是不是最佳的工具。

应用是完成一个特定任务的程序。现代应用通常都涉及某种用户交互。换句话说，应用会由用户驱动，用户指导应用采取某些动作。应用通常有多种功能。例如，用户可以使用一个照片编辑应用完成图像的缩放，绘色或打印。能够成包购买的大多软件都是应用。应用通常是相当大而且往往很复杂的程序。

脚本通常要完成一个任务，或者一组相关的任务。可以有一个脚本自动地对邮件排序，或者备份重要文件。脚本的运行通常无需用户交互，也许会在每天的特定时间完成，或者是因某个事件触发，如收到一个新邮件。有操作系统级的脚本（如每晚会压缩文件的脚本），有应用级的脚本（如自动完成收缩和打印图像的脚本）。自动化是脚本定义中的一个重要部分，脚本通常就是编写代码来自动完成一系列步骤，否则这些步骤就得用户手工完成。

下面，请考虑脚本语言和编程语言之间的差别。并非所有脚本都一定要用脚本语言编写。完全可以用 C 编程语言编写一个对邮件排序的脚本，或者可以使用 Perl 脚本语言编写一个等价的脚本。类似地，并非所有应用都要用编程语言来编写。如果有人想用 Perl 编写一个 Web 浏览器，这也是能做到的。二者之间的界线并不分明。实际上，Perl 语言如此灵活，以至于许多程序员都把它既当作编程语言又当作脚本语言。

最后，最关键的是，要看哪一种语言能提供所需的功能。如果想与操作系统之间大量交互，可能会考虑一个脚本语言，因为脚本语言对操作系统交互提供了更好的支持。如果项目规模很大，而且涉及大量的用户交互，那么从长远看来，采用编程语言可能更容易一些。

#### 实用示例一 加密口令

假设有一个系统，它会把用户看到和键入的所有内容都写到一个文件中，以供审计用。这个文件只能由系统管理员读取，这样在出现问题时，他就能发现究竟谁是罪魁祸首。以下是这个文件的一个片段：

```
Login: bucky-bo
Password: feldspar

bucky-bo> mail

bucky-bo has no mail

bucky-bo> exit
```

一方面系统管理员可能希望保留所有用户行为的一个日志，另一方面他可能还希望把每个人的口令模糊化，以防止黑客以某种方式获得这个文件。对于这个项目，脚本像是一个很自然的选择，因为这应当是自动的，也许要在每天的最后时间完成。不过，项目中有一部分（加密）可能不适于采用脚本语言。加密库主要是针对一些高级语言的，如 C 和 C++。因此，一种可能的实现是编写一个脚本，它会调用一个 C++ 程序来完成加密。

以下脚本使用了 Perl 语言，不过，几乎所有脚本语言都可以完成这个任务。我们之所以选择 Perl，是因为它是跨平台的，而且提供了一些工具可以更容易地完成文本解析。如果你不了解 Perl，也能看得懂这个例子。此例中，Perl 语法最重要的元素就是 ` 字符。` 字符用以指示 Perl 脚本要交付（shell out）给一个外部命令。在这个例子中，脚本会交付给一个名为 encryptString 的 C++ 程序。

这个脚本的策略是循环处理文件的每一行，查找包含口令提示的行。此脚本会写一个新文件 userlog.out，其中包含与源文件相同的文本，只不过这里的所有口令会加密。第一步是打开输入文件来读取，并打开输出文件供写出。然后，脚本需要循环处理文件中的每一行。各行会依次置于一个名为 \$line 的变量中。

```
open (INPUT, "userlog.txt") or die "Couldn't open input file!";
open (OUTPUT, ">userlog.out") or die "Couldn't open output file!";

while ($line = <INPUT>) {
```

接下来，再根据一个正则表达式 (regular expression) 检查当前行，查看该行是否包含 Password 提示。如果确实包含口令提示，Perl 就会把口令存储在变量 \$1 中。

```
if ($line =~ m/^Password: (.*)/) {
```

由于发现了一个匹配，脚本会基于检测到的口令调用 encryptString 程序，得到口令的一个加密版本。此程序的输出保存在 \$result 变量中，程序返回的结果状态码则存储在变量 \$? 中。此脚本会检查 \$?，如果存在问题，则立即退出。如果一切正常，口令行会写至输出文件，在此会以加密的口令取代原口令。

```
$result = `encryptString $1`;
if ($? != 0) { exit(-1) }
print OUTPUT "Password: $result\n";
```

如果当前行不是口令提示，脚本只是原样将这一行写至输出文件，在循环的最后，它会关闭输入和输出文件，并退出。

```
    } else {
        print OUTPUT "$line";
    }
}

close (INPUT);
close (OUTPUT);
```

就这么简单！另外还需要的部分只剩下具体的 C++ 程序了。加密算法的实现超出了本书的范围。重点在于 main() 函数，因为它要接受应加密的字符串作为它的一个参数。

参数包含在 C 风格的字符串数组 argv 中。在访问 argv 的元素之前一定要先查看 argc 参数。要记住，如果 argc 为 1，那么参数表中只有一个元素，它要作为 argv [0] 来访问。argv 数组的第 0 个元素通常就是程序名，所以真正的参数会从 argv [1] 开始。

以下是对输入字符串加密的 C++ 程序的 main() 函数。注意这个程序返回 0 表示成功，返回非 0 表示失败，这是 UNIX 中的标准做法。

```
int main(int argc, char** argv)
{
    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " string-to-be-encrypted" << endl;
        return -1;
    }

    cout << encrypt(argv[1]);

    return 0;
}
```

实际上这个代码中存在一个明显的漏洞。当待加密字符串作为一个命令行参数传递给 C++ 程序时，

其他用户可以通过进程表看到这个字符串。要把信息传递给 C++ 程序，更安全的做法是通过标准输入发送，这正是专业脚本语言所擅长的。

前面已经了解了将 C++ 程序与脚本语言结合是多么的容易，所以可以充分结合这两种语言的能力来完成自己的项目。可以使用一个脚本语言与操作系统交互，并控制脚本的流程，而采用一个传统的编程语言完成困难的工作。

### 18.2.6 C++ 与汇编代码的混合

通常认为 C++ 是一种快速语言，特别是相对于其他面向对象语言而言。不过，在速度摆在最高位置上时，没有哪一种方法能比得上原始汇编代码。应该记得，在编译程序时，它会从高级 C++ 代码转变为低级汇编代码。自动生成的汇编代码对于大多数用途来说已经足够快了。通常还会在所生成的汇编代码上运行优化程序，让它更快一些。不过，尽管编译器取得了诸多进步，水平高的人还是经常会直接编写超过已编译 C++ 代码的汇编代码。

在 C++ 中，许多编译器都使用关键字 `asm` 以允许程序员插入原始的汇编代码。这个关键字是 C++ 标准的一部分，但是其实现则由编译器自定。在大多数编译器中，可以使用 `asm` 在程序中从 C++ 代码降级到汇编代码。

内联汇编在某些应用中非常有用，如 3-D 图形，但是对于大多数程序我们都不推荐这种做法。避免内联汇编代码有许多原因：

- 一旦包括了针对特定平台的原始汇编代码，代码就不能再移植到另一个处理器了。
- 大多数程序员不了解汇编语言，这样就无法修改或维护代码。
- 汇编代码的可读性很差，所以声誉不高。它会影响程序的使用风格。
- 大多数情况下，根本没有必要使用内联汇编代码。如果程序很慢，可以查看是否存在算法问题，或者参考第 17 章提出的另外一些性能建议。

## 18.3 小结

从本章看来，应当可以得出 C++ 是一种灵活的语言。它介于两类语言之间，既没有与特定平台过分绑定，也不是太过高级和通用。在用 C++ 开发代码时可以放心，你不会把自己永远锁定于这种语言。C++ 可以很容易地与其他技术混合，而且有悠久的历史 and 坚实的代码基，这有助于保证即便到了将来它仍是有意義的。



## 第19章 熟练地测试

如果程序员意识到测试是软件开发过程的一部分，这就说明他已经越过了职业生涯的一大障碍。bug并不是偶然出现的。在每个有相当规模的项目中都会发现 bug。如果有一个好的质量保证（quality-assurance，QA）小组，这有非常重要的意义，不过，并不能把测试的重担全都压在 QA 身上。作为程序员，你的职责是编写能够正常工作的代码，并且能够对其测试来证明代码的正确性。

往往会区别白盒测试（white box testing）和黑盒测试（black box testing），白盒测试是指测试者了解程序的内部工作，黑盒测试则是测试程序的功能，而不考虑其实现。这两种测试对于专业质量的项目都很重要。黑盒测试是最基本的方法，因为它通常是对用户行为建模。例如，一个黑盒测试可以检查诸如按钮等界面组件。如果测试者点击了按钮，但什么也没有发生，那么程序中肯定就存在一个 bug。

黑盒测试并不能暴露所有问题。现代程序往往太过庞大，要想点击每一个按钮，提供每一种输入，并完成所有命令组合，这样的仿真往往无法做到。白盒测试很有必要，因为如果在对象或子系统级编写测试，就能更容易地确保测试覆盖面（范围）。与黑盒测试相比，白盒测试通常更容易编写和完成自动化。这一章所强调的主题通常被认为是白盒测试技术，因为程序员可以在开发过程中使用这些技术。

本章先从一个高层次对质量控制进行讨论，包括查看和跟踪 bug 的一些方法。在此介绍之后，提供了单元测试一节，单元测试是最简单也最有用的一种测试。你可以了解到单元测试的理论和具体实践，还将看到一些实战单元测试例子。接下来，我们会介绍更高级的测试，包括集成测试、系统测试和回归测试。最后，本章将为成功测试提供一组提示。

### 19.1 质量控制

大型编程项目很少能圆满完成，即很少达到所有预期目标。无论在开发阶段期间，还是在开发阶段之后，总会发现一些 bug，并要做出修正。应当认识到大家要共同承担质量控制的责任，还要理解 bug 的生命期，这对于很好地参与团队开发是至关重要的。

#### 19.1.1 谁来负责测试

软件开发组织可能有不同的测试方法。在一个小单位中，可能没有专职测试产品的一群人。测试由单个的开发人员负责，或者公司里的所有员工都要伸出援手，尽量在产品发布之前找出问题。在较大的组织中，会有一个专职的质量保证小组根据一组标准来测试产品，评价其质量。不过，测试的某些方面仍然要由开发人员负责。即便是在一些大型组织中，尽管开发人员不参与正式的测试，你仍要认清楚在更大的质量保证过程中你有什么样的责任。

#### 19.1.2 bug 的生命期

所有好的工程小组都认识到，软件发布前和发布后，都会出现 bug。处理这些问题有许多不同的方法。图 19-1 显示了一个正式的 bug 过程，这里用流程图的形式表述。在这个过程中，bug 总是由 QA 小

组中的一个成员提交。bug 报告软件向开发经理发出一个通知，开发经理设置这个 bug 的优先级，并将其分配至相应的模块所有者（编写模块的开发人员）。模块所有者可能接受这个 bug，也可能做出解释，指出这个 bug 实际上属于另一个模块，或者这并不算是个 bug，如果属于另一个模块，开发经理就有机会把这个 bug 分配至别人（另一个模块的所有者）。一旦找到了适当的 bug 所有者，就可以做出修正，开发人员将此 bug 标记为“已修正”（fixed）。此时，QA 工程人员会验证该 bug 不再存在，并把 bug 标记为“关闭”（closed，即清除），或者如果 bug 仍然存在，则再次打开（reopen，即提交）这个 bug（译者注：这里采用“打开”和“关闭”的说法，可以与 bug 生命期对应起来，即 bug 的提交意味着 bug 过程的打开，而 bug 的清除意味着 bug 过程的关闭）。

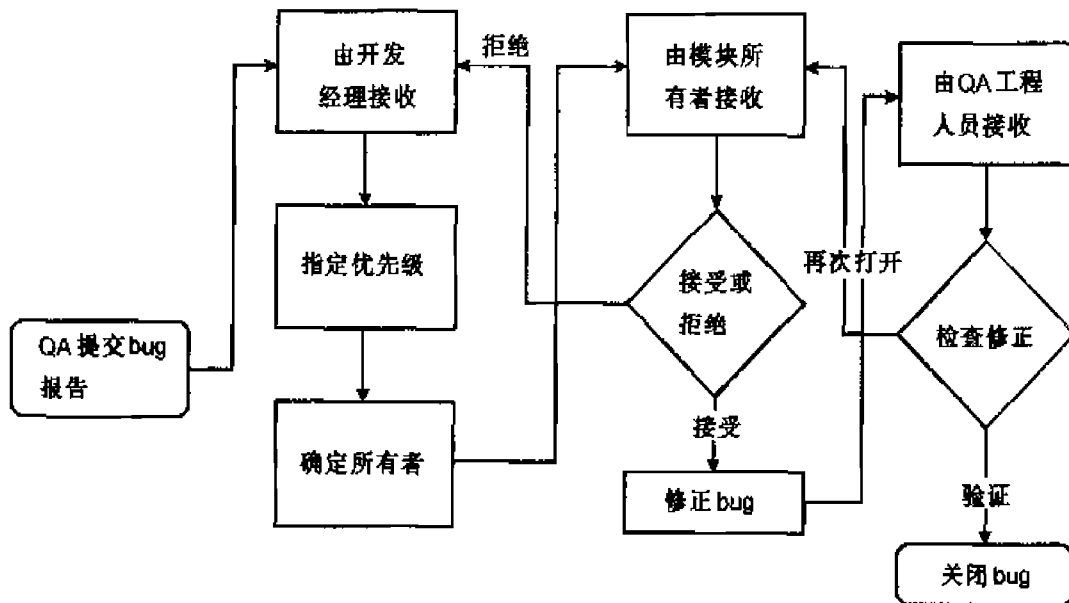


图 19-1

图 19-2 显示了一种不那么正式的方法。在这个工作流中，所有人都可以提交一个 bug，并分配一个初始优先级和模块。模块所有者接收 bug 报告，可能接受，也可能把它再分配到另一个工程人员或模块。做出修正后，bug 会标记为“已修正”。到了测试阶段的最后，所有实现和 QA 工程人员会划分出已修正的 bug，并验证各个 bug 是否不再出现在当前版本中。如果所有 bug 都标记为“关闭”，就表示这个版本可以发布了。

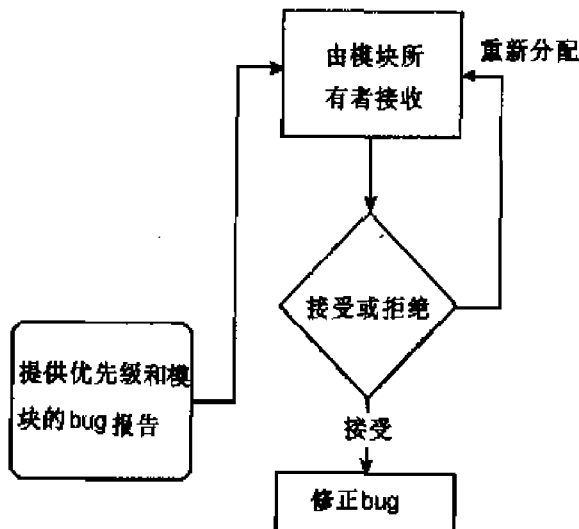


图 19-2

### 19.1.3 bug 跟踪工具

跟踪软件 bug 有许多方法, 从非正式的基于电子邮件或电子表格的机制, 到昂贵的第三方 bug 跟踪软件, 方法相当多。你的组织选择何种解决方案合适, 取决于组织的规模、软件本身、以及你是否希望正式地修正 bug 以及正式程度如何。

Bugzilla 是一个流行的免费工具, 可以用于 bug 跟踪, 这是由编写 Mozilla Web 浏览器的人编写的。作为一个开源项目, Bugzilla 已经逐渐积累了许多有用的特性, 现在甚至能与昂贵的 bug 跟踪软件包相媲美。它的主要特性包括:

- 可定制 bug 设置, 包括其优先级、相关组件、状态等等。
- 能够通过电子邮件通知新的 bug 报告, 或通知对现有报告有修改。
- 能够跟踪 bug 之间的依赖关系, 并能分析得出重复的 bug。
- 提供了报告和搜索工具。
- 提供了一个基于 Web 的界面, 可以提交和更新 bug。

图 19-3 所示为向 Bugzilla 项目输入的一个 bug, 这个 bug 是我们专门为这本书“制造”的。出于我们的目的, 每一章都输入为一个 Bugzilla 组件。提交 bug 的人可以指定 bug 的严重程度 (算不算大问题), 还能指定 bug 的优先级 (需要多快得到修正)。在此还包括一个总结和描述, 基于这个总结和描述, 可以搜索该 bug, 或者以报告格式将其列出。

诸如 Mozilla 等 bug 跟踪工具已经成为专业软件开发环境中一个必不可少的部分。除了提供一个中心列表外 (其中列出了当前打开或提交的 bug), bug 跟踪工具还能提供一个重要的存档记录, 其中记录了原来出现的 bug 和相应的修正。例如, 顾客报告问题后, 支持人员可以使用 Bugzilla 来搜索与之相近的问题。如果已经做出了修正, 支持人员就可以告诉顾客需要更新为哪个版本, 或者告知如何解决问题。

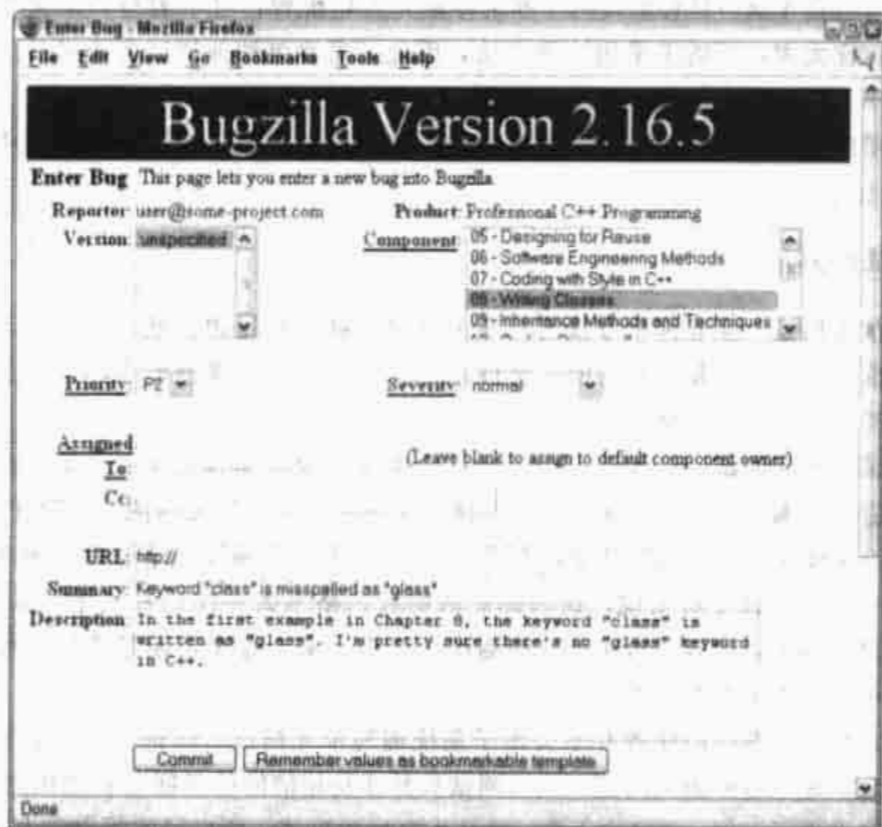


图 19-3

## 19.2 单元测试

要想找到 bug，惟一的途径就是通过测试。在开发人员看来，最重要的一种测试就是单元测试。单元测试是一段用于测试某个类或子系统特定功能的代码。你写得最多的可能就是这种测试。理想情况下，对代码所能完成的每个低级任务都应当存在一个或多个单元测试。例如，假设你在编写一个数学库，它能完成加法和乘法。你提供的单元测试包可能就要包括以下测试：

- 加法基本测试
- 测试大数相加
- 测试负数相加
- 测试 0 与一个数相加
- 测试加法交换律
- 乘法基本测试
- 测试大数相乘
- 测试负数相乘
- 测试乘 0
- 测试乘法交换律

写得好的单元测试可以从多个方面提供保护。第一，可以证明一个功能确实能工作。除非有代码真正地使用你的类，在此之前，并不能知道类的表现如何。第二，如果一个新增加的修改产生了破坏（破坏了原本正常的功能），单元测试会尽早做出警告。这种特殊使用称为回归测试（regression test），本章后面将会介绍。第三，当用作开发过程的一部分时，单元测试要求开发人员从一开始就要着手修正问题。如果代码未能通过单元测试，以至于不允许交付，就必须马上解决问题。第四，在其他代码掺合进来之前，单元测试使你有机会先试一试你的代码。如果你最先开始编程，可以编写一个完整的程序，然后尝试运行。专业程序往往太大，不适于采用这种方法，因此，需要单独地测试各个组件。最后一点，但并不是最次要的是，单元测试可以提供使用例子。几乎可以把这当作一个“副产品”，单元测试可以为其他程序员提供不错的参考代码。如果有同事想知道如何使用你的数学库来完成矩阵相乘，就可以让他参考一个合适的单元测试。

### 19.2.1 单元测试的方法

单元测试一般很难出错，除非你根本没有写单元测试，或者写得很糟糕。一般地，测试越多，所覆盖的代码就越多。覆盖面越大，漏掉 bug 的机会就越少，而且会更少遗憾地告诉你的老板（或者更糟糕地是告诉顾客）：“噢，我们还没测试呢。”

要想最有效地编写单元测试，已经有这样的一些方法了。极限编程方法（见第 6 章的解释）就要求采用这种方法的人应当在编写代码之前编写单元测试。从理论上说，先编写测试有助于明确组件的需求，并提供一个度量，这个度量可用以确定什么时候组件才算完成。先编写测试可能很困难，而且要求程序员必须刻苦钻研。对于一些程序员来说，这与他们的编程风格不太一样。另一种不那么严格的方法是，在编写代码之前先设计测试，但以后再实现这些测试。这样一来，程序员还是要理解模块的需求，不过不必编写代码来使用那些尚不存在的类。

在有些小组中，特定子系统的作者并不为该子系统编写单元测试。理论上讲，如果为自己的代码编写测试，可能会无意识地绕过知道的问题，或者测试只是涉及相信的代码能妥善处理的一些情况。另外，在刚刚编写的代码中发现了 bug，这往往不是一件让人高兴的事情，所以可能不会全力去编写这个单元测试。实践中，让一个开发人员为另一个开发人员的代码编写单元测试，可能需要许多额外的开销和协调。

不过,如果能够达成这种协调,这种方法将有助于保证更有效的测试。

要保证单元测试确实对代码中适当的部分做了测试,另一种方法是,编写单元测试时,尽量扩大代码覆盖面。可以使用一个代码覆盖工具(如 gcov),或者让一个 Perl 高手编写一个脚本,来告诉你单元测试调用的公共方法占多大比例。理论上讲,要对类做适当的测试,那么对于它的所有公共方法都应当有单元测试。

## 19.2.2 单元测试过程

为代码提供单元测试的过程从写代码之前就开始了。即使你不同意先写单元测试后写代码的方法,也应该花些时间考虑你要提供哪些类型的测试。这样一来,就可以把任务分解为定义明确(well-defined,即良定义)的小块,每一块都有自己的测试验证标准。例如,如果你的任务是编写一个数据库访问类,可以先编写将数据插入到数据库中这样一个功能。通过一组单元测试对此做了充分测试之后,就能继续编写支持更新、删除和选择等功能的代码,每写一部分就要做相应的测试。

以下所列的步骤是我们为设计和实现单元测试建议的方法。与所有编程方法一样,最好的过程应当是能够得到最好结果的过程。建议采用不同的方法来使用单元测试,以确定哪一种最适合。

### 定义测试的粒度

在开始设计单个的测试之前,需要做一个客观的实际检查。给定组件的需求、组件的复杂性,以及有多少时间来完成单元测试,能提供何种层次的单元测试呢?在一个理想世界中,除了充分地验证程序的功能外,还需要编写更多的测试(不过如果真的是理想世界,我们可能根本不需要测试了,因为一切都会正常工作!)。在实际中,你的时间可能已经很紧了,起码的任务就是在给定约束条件的情况下尽量增加单元测试的有效性。

测试的粒度(granularity)是指测试的范围。如表 19-1 所示,可以只通过几个测试函数对一个数据库类进行单元测试,也可以将其“一网打尽”,完全确保一切都工作正常。

表 19-1

粗粒度测试	中粒度测试	细粒度测试
testConnection()	[所有粗粒度测试]	[所有粗粒度和中粒度测试]
testInsert()	testConnectionDroppedError()	testConnectionThroughHTTP()
testUpdate()	testInsertBadData()	testConnectionLocal()
testDelete()	testInsertStrings()	testConnectionErrorBadHost()
testSelect()	testInsertIntegers()	testConnectionErrorServerBusy()
	testUpdateStrings()	testInsertWideCharacters()
	testUpdateIntegers()	testInsertLargeData()
	testDeleteNonexistentRow()	testInsertMalformed()
	testSelectComplicated()	testUpdateWideCharacters()
	testSelectMalformed()	testUpdateLargeData()
		testUpdateMalformed() test
		DeleteWithoutPermissions()
		testDeleteThenUpdate()
		testSelectNested()
		testSelectWideCharacters()
		testSelectLargeData()

可以看到，后面的列都比前面的列提供了更多测试。从粗粒度测试转向更细粒度的测试时，要考虑错误条件、不同的输入数据集和不同的操作模式。

当然，在选择测试的粒度时，最初所做的决定并不一定铁定不变。也许这个数据库类只是用来说明有关概念的，可能根本不会实际使用。如此只需几个简单的测试可能就足够了，可以以后再增加更多测试。或者，也许以后哪一天用例有所修改。编写数据库类时可能没有考虑到国际化字符。一旦增加了这样的一些特性，就应当利用特定的目标单元测试来进行测试。

如果你想以后再来查看或改进测试，这是很正确的想法，一定要尽量这样做。要把单元测试看作是实际实现的一部分。做出修改时，不要只是修改原来的测试，让它能继续工作就行了，这是不够的，还应当编写新的测试，而且要重新评价现有的测试。

单元测试是所测试子系统的一部分。在改进子系统的同时，也应当改进测试。

### 头脑风暴考虑单个测试

经过一段时间，你可能会有一种直觉，意识到代码的某些方面应当拿来进行单元测试。某些方法或输入让人感觉就是应该测试。这种直觉来自于尝试和错误，另外通过查看小组中其他人编写的单元测试也能让人有这种感受。哪些程序员是最好的单元测试人员，这很容易分辨出来。这些人的测试往往组织得当，而且会经常修改。

创建单元测试要成为一种习惯，在此之前，先要完成一个任务，明确哪些测试要通过头脑风暴的方式编写。为了对此有些认识，请考虑以下问题。

1. 写这段代码是要做什么？
2. 各个方法通常会以哪些典型方式调用？
3. 调用者可能违反方法的哪些前置条件？
4. 各个方法可能会如何遭到误用？
5. 希望哪些类型的数据作为输入？
6. 不希望哪些类型的数据作为输入？
7. 有哪些边缘情况或异常条件？

不必为这些问题给出正式的答案（除非你的经理非常推崇这本书），不过这些问题应当有助于你对单元测试有所认识。数据库类的测试表所包含的测试函数就是针对上述问题得出的。

在对想要使用的测试有了想法之后，可以考虑如何把这些测试分类，而且各类测试应该按顺序进行。在数据库类例子中，测试应当分为以下各类：

- 基本测试
- 错误测试
- 国际化测试
- 错误输入测试
- 复杂测试

对测试分类可以更容易明确和扩展，而且还能更容易地了解代码的哪些方面已经得到了充分测试，而哪些方面还需要使用更多的单元测试。

编写大量简单测试很简单，但是不要忘记更复杂的情况！

### 创建示例数据和结果

在编写单元测试时，最常遇到的陷阱就是让测试与代码行为匹配，而不是使用测试来验证代码。如



果编写了一个单元测试，它要完成一个数据库选择，而这个测试失败了，这是代码的问题还是测试的问题呢？通常很容易认为代码是正确的，并修改测试来满足代码。这种做法往往是错误的。

为了避免这个陷阱，应当在尝试测试之前，先了解测试的输入和预期的输出。有时候说起来比做起来简单。例如，假设编写了一些代码，可以使用一个特定密钥对一个任意的文本块加密。一个合理的单元测试会取一个固定的文本串，并把它传递至加密模块。然后，它会检查结果，查看是否得到了正确的加密。在编写这样一个测试时，往往会先尝试得出加密模块的行为，并查看结果。如果看上去是合理的，可能会为此编写一个测试来寻找这个结果值。不过，这样做并不能证明什么。你并没有真正测试代码，只是编写了一个测试来保证能够继续重复地返回同样的值而已。编写测试通常需要做一些实际的工作，要脱离原来的加密模块独立地对文本加密，来得到一个正确的结果。

在运行测试之前，先确定测试的正确输出。

### 编写测试

取决于当前采用的是何种类型的测试框架，测试的实际代码可能有所不同。以下将讨论一个测试框架 `cppunit`。不过，以下原则将有助于保证有效的测试，这些原则独立于实际的实现：

- 确保每个测试中只对一个方面进行测试。这样一来，如果一个测试失败，就能指出哪个特定的功能出了问题。
- 测试要具体。测试失败是因为抛出一个异常还是因为返回了错误的值？
- 在测试代码中大量使用日志。如果某一天测试失败了，就可以通过日志了解到底发生了什么。
- 避免测试依赖于前期的测试，还要避免相互关联的测试。测试应当尽可能地具备原子性和独立性。
- 如果测试需要使用其他子系统，可以考虑为这些子系统编写桩版本（`stub version`），使之能模拟模块的行为，这样如果与之关联不大的代码有修改，也不会导致测试失败。
- 要求代码审查人员还要审查你的单元测试。在完成一个代码审查时，告诉其他人员你觉得哪里还需要增加另外的测试。

从以下例子中可以看到，单元测试通常是非常小而且非常简单的代码段。在大多数情况下，编写一个单元测试只需要几分钟而已，因此，单元测试往往是最经常写的一种测试。

### 运行测试

写完一个测试时，应当马上运行，而不要让人为了得到结果而等得太焦急。如果能看到一个完全通过单元测试的屏幕，这种喜悦是无以言表的。对于大多数程序员来说，要提供量化数据来证明代码是有用而且正确的，这也是一种最简单的方法。

即使采用了先写测试再写代码的方法，也应当写完测试之后就立即运行。这样一来，就可以证明测试最初确实会失败。等到有实际代码之后，就有了具体的数据可以显示它能达到预期的目的。

要想让每个测试第一次都能得到预期的结果，这是不太可能的。理论上讲，如果在编写代码之前写了测试，所有测试都应该失败。如果某个测试通过了，要么是因为魔法般地出现了实际代码，要么是因为测试本身存在问题。如果已经完成了代码，但是测试仍然失败（有人可能会说，如果测试失败，那么代码并没有真正完成！），这就有两种可能。代码可能出错，也可能测试出错。前面已经提到，通常会考虑去修改测试，调整几个布尔值来让一切运转起来。一定要抵制这种诱惑！

### 19.2.3 实战单元测试

前面已经从理论上了解了单元测试，下面可以编写一些实际的测试了。以下例子取自第 17 章的对象池示例。先简单回顾一下，对象池是一个可以用于避免过度创建对象的类。通过跟踪已经创建的对象，

对象池相当于一个中间人，它介于需要某类对象的代码和已经存在的此类对象之间。ObjectPool 的公共接口如下所示：

```
//
// template class ObjectPool
//
// Provides an object pool that can be used with any class that provides a
// default constructor.
//
// The object pool constructor creates a pool of objects, which it hands out
// to clients when requested via the acquireObject() method. When a client is
// finished with the object it calls releaseObject() to put the object back
// into the object pool.
//
// The constructor and destructor on each object in the pool will be called only
// once each for the lifetime of the program, not once per acquisition and release.
//
// The primary use of an object pool is to avoid creating and deleting objects
// repeatedly. The object pool is most suited to applications that use large
// numbers of objects for short periods of time.
//
// For efficiency, the object pool doesn't perform sanity checks.
// Expects the user to release every acquired object exactly once.
// Expects the user to avoid using any objects that he or she has released.
//
// Expects the user not to delete the object pool until every object
// that was acquired has been released. Deleting the object pool invalidates
// any objects that the user had acquired, even if they had not yet been released.
//
template <typename T>
class ObjectPool
{
public:
    //
    // Creates an object pool with chunkSize objects.
    // Whenever the object pool runs out of objects, chunkSize
    // more objects will be added to the pool. The pool only grows:
    // objects are never removed from the pool (freed), until
    // the pool is destroyed.
    //
    // Throws invalid_argument if chunkSize is <= 0.
    //
    ObjectPool(int chunkSize = kDefaultChunkSize)
        throw(std::invalid_argument, std::bad_alloc);

    //
    // Frees all the allocated objects. Invalidates any objects that have
    // been acquired for use.
    //
    ~ObjectPool();

    //
    // Reserve an object for use. The reference to the object is invalidated
    // if the object pool itself is freed.
    //
    // Clients must not free the object!
```

```
//
T& acquireObject();

//
// Return the object to the pool. Clients must not use the object after
// it has been returned to the pool.
//
void releaseObject(T& obj);

// [Private/Protected methods and data omitted]
};
```

如果还不了解对象池的概念，在继续学习这个例子之前，可以先回过头去复习第 17 章。

### cppunit 介绍

cppunit 是面向 C++ 的一个开源单元测试框架，它基于一个名为 junit 的 Java 包。这个框架相当轻量级（这是一个优点），而且很容易着手使用。使用诸如 cppunit 测试框架的优点在于，开发人员可以把重点放在编写测试上，而不用过多地处理建立测试、构建测试有关逻辑以及收集结果等工作。cppunit 为测试开发人员提供了许多有用的实用工具，还能以多种不同格式提供自动化输出。这里并不打算完备地介绍 cppunit 的特性。建议访问 <http://cppunit.sourceforge.net> 来了解有关 cppunit 的内容。

使用 cppunit 最常用的方法就是派生 CppUnit::TestFixture 类的子类（注意，CppUnit 是命名空间，TestFixture 是类）。测试组（fixture）就是逻辑上的一组测试。TestFixture 子类可以覆盖 setUp() 方法来完成测试运行前需要做的任务，还可以覆盖 tearDown() 方法，这个方法可以用于在测试运行完后做清除工作。测试组还可以用成员变量维护状态。

objectPoolTest 的骨架实现如下，这是一个用于测试 objectPool 类的类：

```
// ObjectPoolTest.h

#include <cppunit/TestFixture.h>

class ObjectPoolTest : public CppUnit::TestFixture
{
public:
    void setUp();
    void tearDown();
};
```

由于对 ObjectPool 的测试相当简单，而且很独立，所以 setUp() 和 tearDown() 的定义都为空就足够了。源文件最初可以如下所示：

```
// ObjectPoolTest.cpp

#include "ObjectPoolTest.h"

void ObjectPoolTest::setUp()
{
}

void ObjectPoolTest::tearDown()
{
}
```



学习在线

视频资料下载  
电子书交流

[www.eimhe.com](http://www.eimhe.com)

这就是开发单元测试的全部初始代码！

### 编写第一个测试

由于这可能是你第一次接触 cppunit，或者可能是第一次接触单元测试，所以第一个测试将是一个非常简单的测试。它只是测试 0 是否小于 1。

cppunit 中的单个单元测试只是测试组类的一个方法。要创建一个简单测试，需要将这个方法的声明增加到 ObjectPoolTest.h 文件中：

```
// ObjectPoolTest.h

#include <cppunit/TestFixture.h>

class ObjectPoolTest : public CppUnit::TestFixture
{
public:
    void setUp();
    void tearDown();

    // Our first test!
    void testSimple();
};
```

这个测试定义使用了 CPPUNIT\_ASSERT 宏来完成实际的测试。CPPUNIT\_ASSERT 与用过的其他断言宏一样，只是把应当正确的一个表达式包围起来。有关断言的详细内容请参见第 20 章。在这个例子中，测试认为 0 小于 1，因此它把语句  $0 < 1$  包围在一个 CPPUNIT\_ASSERT 宏调用中。这个宏定义在 cppunit/TestAssert.h 文件中。

```
// ObjectPoolTest.cpp

#include "ObjectPoolTest.h"
#include <cppunit/TestAssert.h>

void ObjectPoolTest::setUp()
{
}

void ObjectPoolTest::tearDown()
{
}

void ObjectPoolTest::testSimple()
{
    CPPUNIT_ASSERT(0 < 1);
}
```

如此而已！当然，大多数单元测试还会做一些更有意思的工作，而不只是一个简单的断言。可以看到，常用的模式就是完成某种计算，并断言结果就是预想的值。利用 cppunit，甚至不必考虑异常，框架会根据需要捕获和报告异常。

### 构建一组测试

在这个简单测试实际运行之前，还有几步要做。cppunit 把一组测试作为一个测试套件 (suite) 来运行。测试套件告诉 cppunit 要运行哪些测试，这与测试组不同，测试组只是把测试逻辑地分组。

常用模式是为测试组类提供一个静态方法，此方法构建一个测试套件，其中包含所有测试。在更新后的 ObjectPoolTest.h 和 ObjectPoolTest.cpp 中，suite() 方法就是做此使用。

```
// ObjectPoolTest.h
```

```
#include <cppunit/TestFixture.h>
```

```
#include <cppunit/TestSuite.h>
```

```
#include <cppunit/Test.h>
```

```
class ObjectPoolTest : public CppUnit::TestFixture
```

```
{
```

```
    public:
```

```
        void setUp();
```

```
        void tearDown();
```

```
        // Our first test!
```

```
        void testSimple();
```

```
        static CppUnit::Test* suite();
```

```
};
```

```
// ObjectPoolTest.cpp
```

```
#include "ObjectPoolTest.h"
```

```
#include <cppunit/TestAssert.h>
```

```
void ObjectPoolTest::setUp()
```

```
{
```

```
}
```

```
void ObjectPoolTest::tearDown()
```

```
{
```

```
}
```

```
void ObjectPoolTest::testSimple()
```

```
{
```

```
    CPPUNIT_ASSERT(0 < 1);
```

```
}
```

```
CppUnit::Test* ObjectPoolTest::suite()
```

```
    CppUnit::TestSuite* suiteOfTests = new CppUnit::TestSuite("ObjectPoolTest");
```

```
    suiteOfTests->addTest(new CppUnit::TestCaller<ObjectPoolTest>{
```

```
        "testSimple",
```

```
        &ObjectPoolTest::testSimple } );
```

```
    return suiteOfTests; // Note that Test is a superclass of TestSuite
```

```
}
```

创建 TestCaller 的模板语法有些复杂，不过因为几乎所写的每个测试都会遵循这种模式，所以可以忽略 TestSuite 和 TestCaller 实现的绝大部分。

为了真正运行这个测试套件，并查看结果，需要一个测试运行器 (test runner)。cppunit 是一个灵活的框架。它包含了许多不同的运行器，可以在不同的环境中运行，如 MFC Runner 就设计为在一个利用



Microsoft Foundation Classes (MFC 基类库) 编写的程序中运行。对于一个基于文本的环境, 应当使用 Text Runner, 这个运行器在 CppUnit::TextUi 命名空间中定义。

运行测试套件的代码由以下 ObjectPoolTest 测试组定义。它只是创建一个运行器, 增加 suite() 方法返回的测试, 并调用 run()。

```
// main.cpp

#include "ObjectPoolTest.h"
#include <cppunit/ui/text/TestRunner.h>

int main(int argc, char** argv)
{
    CppUnit::TextUi::TestRunner runner;
    runner.addTest(ObjectPoolTest::suite());
    runner.run();
}
```

代码经过编译、链接和运行后, 应该能看到类似下面的输出:

```
OK (1 tests)
```

如果修改代码, 断言  $1 < 0$ , 测试将失败, cppunit 会如下报告失败:

```
!!!FAILURES!!!
Test Results:
Run: 1 Failures: 1 Errors: 0

1) test: testSimple (F) line: 21 ObjectPoolTest.cpp
assertion failed
- Expression: 1 < 0
```

需要注意, 通过使用 CPPUNIT\_ASSERT 宏, 此框架能够准确地指出测试在哪一行上失败了, 这对于调试来说是一个很有用的信息。

### 增加真实的测试

既然框架已经建好, 而且也成功运行了一个简单测试, 下面要把注意力转向 ObjectPool 类, 并编写一些代码对它真正做些测试。类似于上述简单测试, 以下所有测试都要增加到 ObjectPoolTest.h 和 ObjectPoolTest.cpp。

在写测试之前, 需要一个辅助对象与 ObjectPool 类一同使用。应该记得, ObjectPool 类会创建成块的某类对象 (一块中包括许多对象), 并根据调用者的请求将对象交给调用者。需要有一些测试来检查所获取的对象是否等同于先前已经获取的对象。为此, 一种办法是创建一个序列对象的对象池, 其中的对象都有一个单调递增的序列号。以下代码定义了这个类:

```
// Serial.h

class Serial
{
public:
    Serial();

    int getSerialNumber() const;
```

```
protected:
    static int sNextSerial;

    int mSerialNumber;
};

// Serial.cpp

#include "Serial.h"
Serial::Serial()
{
    // A new object gets the next serial number.
    mSerialNumber = sNextSerial++;
}

int Serial::getSerialNumber() const
{
    return mSerialNumber;
}

int Serial::sNextSerial = 0; // The first serial number is 0.
```

下面开始完成测试。作为最初的正常检查，可能希望有这样一个测试，它只是创建一个对象池。如果在创建过程中抛出任何异常，cppunit 就会报告一个错误：

```
void ObjectPoolTest::testCreation()
{
    ObjectPool<Serial> myPool;
}
```

下一个测试是一个消极测试(negative test)，因为它会做一些应当失败的事情。在这个例子中，此测试试图创建一个块大小非法(为 0)的对象池。对象池构造函数应当抛出一个异常。正常情况下，cppunit 应当捕获这个异常，并报告一个错误。不过，由于这是我们所期望的行为，因此测试会显式捕获异常，并设置一个标志。测试的最后一步是断言该标志已设置。因此，如果构造函数没有抛出异常，测试将失败。

```
void ObjectPoolTest::testInvalidChunkSize()
{
    bool caughtException = false;

    try {
        ObjectPool<Serial> myPool(0);
    } catch (const invalid_argument& ex) {
        // OK. We were expecting an exception.
        caughtException = true;
    }

    CPPUNIT_ASSERT(caughtException);
}
```

testAcquire() 测试了一个特定的公共功能，即 ObjectPool 交出对象的能力。在这个例子中，没有什么可以加断言。为了证明所得到 Serial 引用的合法性，此测试会断言序列号大于或等于 0。

```
void ObjectPoolTest::testAcquire()
{
    ObjectPool<Serial> myPool;

    Serial& serial = myPool.acquireObject();

    CPPUNIT_ASSERT(serial.getSerialNumber() >= 0);
}
```

下一个测试更有意思。ObjectPool 不能把同一个 Serial 对象交出两次（除非已经显式释放）。这个测试会创建一个固定块大小的对象池，再获取对象，而且获取的对象个数恰好等于块大小，由此检查 ObjectPool 的惟一性。如果对象池适当地分发了惟一的对象，这些对象的序列号就不会相同。需要注意，这个测试只考虑了在一块内创建的对象。也可以对多块做一个类似的测试，这是一个不错的想法。

```
void ObjectPoolTest::testExclusivity()
{
    const int poolSize = 5;
    ObjectPool<Serial> myPool(poolSize);
    set<int> seenSerials;

    for (int i = 0; i < poolSize; i++) {
        Serial& nextSerial = myPool.acquireObject();

        // Assert that this number hasn't been seen before.
        CPPUNIT_ASSERT(seenSerials.find(nextSerial.getSerialNumber()) ==
                        seenSerials.end());

        // Add this number to the set.
        seenSerials.insert(nextSerial.getSerialNumber());
    }
}
```

这个实现使用了 STL 中的 set 容器。如果对这个容器不太熟悉，有关详细内容请参见第 21 章。

最后一个测试（就目前而言）将检查释放功能。一旦释放一个对象，ObjectPool 就可以将其再次交出。在对象池再次用完所有已释放对象之前，它不能另外创建对象块。这个测试首先从对象池获取一个 Serial，并记录它的序列号。接下来，这个对象立即释放回对象池。然后，从对象池获取对象，直到重新利用到原来这个对象（由其序列号标识），或者整个块已经用完。如果代码找完了整个块，而没有发现再利用的对象（即没有利用到原来释放的这个对象），这个测试就会失败。

```
void ObjectPoolTest::testRelease()
{
    const int poolSize = 5;
    ObjectPool<Serial> myPool(poolSize);

    Serial& originalSerial = myPool.acquireObject();

    int originalSerialNumber = originalSerial.getSerialNumber();

    // Return the original object to the pool.
    myPool.releaseObject(originalSerial);

    // Now make sure that the original object is recycled before
    // a new chunk is created.
```

```

bool wasRecycled = false;
for (int i = 0; i < poolSize; i++) {
    Serial& nextSerial = myPool.acquireObject();
    if (nextSerial.getSerialNumber() == originalSerialNumber) {
        wasRecycled = true;
        break;
    }
}

CPPUNIT_ASSERT(wasRecycled);
}

```

这些测试增加到测试套件中后，就能投入运行，而且它们都应当能通过。当然，如果一个或多个测试失败，就会面对单元测试中的经典问题：是测试有问题，还是代码有问题？

### 收集单元测试结果

我们希望，通过前几节的测试，能让你对如何为实际代码编写真正专业质量的测试有一个很好的认识。不过，这还只是冰山之一角。前面的例子应当有助于你考虑是否还可以为 ObjectPool 类编写其他的测试。例如，这些测试都没有处理分配多块的情况，而无疑这也是需要覆盖的功能。而且在此没有多次请求和释放同一对象的复杂测试。对于一段给定代码，能写多少单元测试是没有限制的，这也是单元测试最妙的地方！如果想知道代码针对某种情况会有何种反应，那就写个单元测试吧！如果子系统的某个方面看上去有问题，就要扩大该领域的单元测试覆盖范围。即使只是想从客户的角度了解一下如何使用类，编写单元测试也是获得其他观点的一个好办法。

## 19.3 高级测试

单元测试是防止 bug 的最好的第一道防线，但它们还只是测试过程的一部分，测试过程往往比单元测试大得多。高级测试强调的是产品的各部分如何协同工作，而不像单元测试那样只强调片面。从某种意义上说，高级测试更难编写，因为需要编写哪些测试往往不太清楚。不过，如果没有对程序各部分如何协同工作做出测试，就不能声称程序确实能工作。

### 19.3.1 集成测试

集成测试（integration test）所覆盖的领域特别适合于组件。单元测试一般只是在一个类的层次上完成测试，不同于单元测试，集成测试往往涉及两个或多个类。集成测试的长处在于测试两个组件之间的交互，这两个组件通常是由两个不同的程序员编写。实际上，编写一个集成测试的过程通常会暴露出设计中的一些重要缺陷。

#### 集成测试示例

由于没有可靠快捷的规则来确定应当编写哪些集成测试，通过一些例子来说明可能有助于你有所认识，能够了解到对于何种情况集成测试是有用的。以下展示了适合集成测试的一些情况，不过这里并没有覆盖所有可能的情况。正如单元测试一样，经过一段时间后，你会对哪些集成测试有用有全新的认识。

#### 基于 XML 的文件串行化工具

假设你的项目包括一个持久存储层，要用于将某些类型的对象保存到磁盘，以及从磁盘将其重新读回。串行化数据时髦方法是使用 XML 格式，这样，组件的逻辑划分可能包括一个 XML 转换层，它位于一个定制文件 API 之上。这些组件可以得到充分的单元测试。XML 层会有一些单元测试，来确保不同类型的对象会正确地转换为 XML，以及能够根据 XML 适当地填充对象。文件 API 也已经有一些测试，

能够测试磁盘上的读、写、更新和删除文件。当这些模块开始一同工作时，就很适合进行集成测试。至少，应当有一个集成测试来测试以下工作：通过 XML 层将一个对象保存到磁盘上，再将其读回，再与原对象做一个比较。由于这个测试覆盖了写和读两个模块，所以这是一个基本的集成测试。

#### 共享资源的阅读器和书写器

假设有一个程序包含一个由多个组件共享的数据空间。例如，股票交易程序可能有一个买单和卖单队列。与接收股票交易请求相关的组件会向队列增加订单，而与完成股票交易相关的组件会从队列中取数据。可以对队列类完成单元测试，但是必须利用实际组件对该队列类的使用进行测试，否则在此之前，并不能知道假设是否有误。一个好的集成测试应当使用股票请求组件和股票交易组件作为队列类的客户。可以编写一些示例订单，并确保它们能够通过客户组件成功地进入和退出队列。

#### 第三方库的包装器

集成测试并不一定总在集成你自己的代码时才出现。许多情况下，还要编写集成测试来测试你的代码与一个第三方库之间的交互。例如，你可能在使用一个数据库连接库，从而与一个关系数据库系统通信。你可能为这个库建立了一个面向对象包装器，以增加对连接缓存的支持，或者提供一个更友好的接口。这就是一个需要测试的非常重要的集成点，因为，即使这个包装器为数据库提供了一个更有用的接口，也有可能引入对原库的误用。换句话说，编写包装器是件好事，但是编写的包装器引入 bug 往往就会带来灾难。

#### 集成测试方法

真正开始编写集成测试时，在集成测试和单元测试之间往往没有明显的界线。如果一个单元测试修改后会涉及另一个组件，它会突然变成一个集成测试吗？在某种意义上，这个答案是什么并没有意义，因为不论测试是何种类型，好测试就是好测试。建议把集成测试和单元测试的概念用作为测试的两种方法，而不要过分考虑每个测试究竟应当属于哪一类。

从实现上讲，集成测试通常使用一个单元测试框架来编写，这就使二者的区别更为模糊。我们已经了解到，单元测试框架提供了一种简单的方法来编写一个 yes/no 测试，并能生成有用的结果。不论测试所考查的是一个功能单元，还是两个组件的交叉，从框架的角度看是很难做出区别的。

出于性能和组织上的原因，可能想把单元测试与集成测试分开。例如，你的小组可能决定每个人在交付新代码之前都必须运行集成测试，不过是否运行与之无关的单元测试则没有这么严格的要求。将两类测试分开还有助于提高结果的价值。如果在 XML 类测试中出现失败，显然这是该类中的一个 bug，而不是该类与文件 API 之间的交互出了问题。

### 19.3.2 系统测试

系统测试 (System test) 比集成测试的层次更高。这些测试会把程序作为一个整体来检查。系统测试通常使用一个虚用户 (virtual user) 来模拟使用此程序的人。当然，虚用户必须编写为一个脚本，它会完成所需的动作。其他系统测试会依赖于脚本或一组固定的输入和预期的输出。

与单元测试和集成测试相似，各个系统测试都会完成一个特定的测试，并希望得到特定的结果。往往会使用系统测试来确保不同特性确实能正常地相互结合，这种做法并不算少。理论上讲，得到充分系统测试的程序会包含一个完备的测试，即针对每一个特性的每一种排列都有考虑。这种方法会很快变得很笨拙，不过还是应该尽力对多种特性的结合使用进行测试。例如，一个图形程序可以有这样一个系统测试，它会导入一个图像、对其放置、完成模糊滤镜、转换为黑白图像，然后保存。这个测试会把所保存的图像与包含有预期结果的文件加以比较。

遗憾的是，有关系统测试的规则很少，因为系统测试相当依赖于实际的应用。对于处理文件而无用



户交互的应用，编写系统测试可能与编写单元测试和集成测试很相似。对于图形程序，可能最好采用虚用户方法。对于服务器，可能需要构建桩客户来模拟网络业务流。重点在于，你要测试程序的实际使用，而不只是程序的一个部分。

### 19.3.3 回归测试

回归测试 (Regression testing) 与其说是一种特定的测试，不如说是一个测试概念。其思想是，一个特性一旦能正常工作，开发人员就可能会放心地将其放在一边，认为它一直都能正常工作。遗憾的是，由于新特性的加入，再加上其他代码的修改，这些加在一起往往会导致原先正常的功能不再正常。回归测试通常会作为一种对已完成和已正常工作的特性所做的正常检查。如果回归测试编写得好，新引入的修改破坏了原来的特性时，回归测试就会停止。如果你的公司在质量保证测试上有充足的人力，回归测试可能会采用手工测试的方式。测试人员相当于用户，他们会完成一系列的步骤，逐步地测试前一个版本中能正常工作的各个特性。如果妥善完成，这种方法很周全而且很准确，不过可扩展性不太好。作为另一个极端，你可能建立了一个完全自动化的系统，可以作为一个虚用户完成每个函数。这此可能要编写一个复杂的脚本，这会很困难，不过已经有许多商业和非商业包可以简化对不同类型应用编写脚本的工作。介于这二者之间，还有一种烟雾测试 (smoke testing)。有些测试只是对应正常工作的一部分特性进行测试，即其中最重要的特性子集。其思想是如果有问题，就会马上反映出来。如果通过了烟雾测试，随后可以完成更严格的手工或自动测试。

有些 bug 就像恶梦一般，不仅很可怕，还会反复出现。反复出现的 bug 很让人灰心，而且没有充分地利用工程资源。出于某些原因，即使决定并不编写一组回归测试，还是应该为已经修正的 bug 编写回归测试。通过对 bug 修正来编写测试，一方面可以证明该 bug 确实已经修正，另一方面还可以建立一个警报器，如果 bug 又出现就会触发这个警报器（例如，如果修改回滚，或者未完成）。如果对原先修正过的 bug 做回归测试时失败了，应当很容易修正，因为回归测试可以指出原来的 bug 编号，并说明第一次是怎么修正的。

## 19.4 成功测试的提示

作为一个软件工程人员，你在测试中的角色可能只是负责基本的单元测试，也可能要统管一个自动化测试系统。由于测试的角色和方式有如此大的差异，我们根据自己的经验总结了一些技巧，希望在各种测试情况下能够对你有所帮助。

- 花些时间来设计你的自动化测试系统。每天都不断运行的系统会很快检测到失败。如果出现失败，系统能向工程人员自动地发出电子邮件，或者发出大音量的鸣叫，就能增加问题的“可见度”。
- 不要忘记压力测试。即使数据库访问类完全通过了整个单元测试套件，但是如果同时要由数十个线程使用，仍有可能出问题。应当在现实世界中可能遇到的最极限条件下测试产品。
- 在不同平台上测试，或者在一个与客户系统所用平台最接近的平台上测试。要在多个操作系统上测试，一种方法是使用一个第三方虚拟机环境，这样可以在同一个机器上运行多种不同的操作系统。
- 可以编写一些测试，有意地在系统中注入错误。例如，可以编写一个正在读取文件时删除文件的测试，或者模拟在网络操作过程中网络瘫痪的情况。
- bug 和测试是紧密相关的，bug 修正应当通过编写一个回归测试来证明。这个回归测试的注释应当指示出原来的 bug 编号。
- 不要只是把失败的测试注释掉。如果一位同事遭遇一个 bug，并发现你注释掉了与之相关的测试，他会来找你麻烦的！



给你一个最重要的提示：要记住测试是软件开发的一部分。如果同意这种观点，而在开始编写代码前就遵循这种观点，那么你应当很清楚地认识到，完成一个特性时，还要做更多工作来证明它确实正常。

## 19.5 小结

本章介绍了有关测试的基本知识，这是所有专业程序员都应该知道的。特定的单元测试是改善代码质量的最简单、最有效的方法。高级测试则会覆盖用例、模块间的同步和回归保护。无论你在测试中处于什么角色，通过本章的学习，现在都应当能够自信地设计、创建和查看不同层次的测试了。

既然已经知道如何找到 bug，下面该学习如何修正这些 bug。为此，第 20 章将介绍有效调试的技术和策略。

## 第20章 征服调试

代码中出现 bug 是在所难免的。每个专业程序员都想编写出没有 bug 的代码，但是事实是，很少有软件工程师在这方面的努力能够奏效。计算机用户都知道，bug 在计算机软件中可谓一种流行病。你写的软件可能也不例外。因此，除非你计划由你的同事负责修正所有 bug，否则如果不知道如何调试 C++ 代码，就不可能成为一名专业的 C++ 程序员。通常有经验的程序员和新手的一大区别就是看他有没有高超的调试技能。

尽管调试的重要性显而易见，但是在课程中和书本上对调试却往往未加足够的重视。调试看起来是一种每个人都想了解的技能，但是没有人知道该怎么来教。这一章会奉上一些具体的调试原则和技术，甚至可以用来调试最棘手的问题。本章内容包括基本调试法则和 bug 分类的介绍，随后会提供一些避免 bug 的提示。对 bug 做出规划的技术包括错误日志、调试轨迹和加断言。接下来会对调试出现的问题提供一些特定的提示，包括再生 bug、调试可再生 bug、调试不可再生 bug、调试内存错误和调试多线程程序等技术。本章的最后会提供一个循序渐进的调试例子。

### 20.1 调试基本法则

调试的第一条规则是要诚实，承认自己的程序可能包含 bug。这种真实的评价能使你尽最大努力避免 bug 在程序中丛生，而且同时能够加入必要的特性，从而使调试尽可能地容易。

调试基本法则：编码时要避免 bug，不过要对代码中的 bug 做出规划。

### 20.2 bug 分类

计算机程序中的 bug 是不正确的运行行为。这种不受欢迎的行为包括灾难性的 bug (catastrophic bug) 和非灾难性的 bug (noncatastrophic bug)，灾难性的 bug 会导致程序终止、数据破坏、操作系统瘫痪或其他一些类似的严重后果；非灾难性的 bug 会导致程序有不正确的表现，但程度要轻得多。例如，一个 Web 浏览器可能会返回不正确的 Web 网页，或者一个电子表格应用可能错误地计算了某一列的标准差。bug 的底层原因（或根本原因）是程序中的错误，正是这些错误导致了这种不正确的行为。调试程序的过程包括决定 bug 的根本原因，以及修正代码以使此 bug 不再出现。

### 20.3 避免 bug

C++ 拥有诸多功能强大的特性，这也使它成为一种极其容易出错的语言，因此与使用大多数其他语言相比，调试技术对于用 C++ 编写代码甚至更为重要。以下是避免程序中出现 bug 的一些提示：

- 通读这本书。深入地了解 C++ 语言，特别是指针和内存管理。然后向你的朋友和同事推荐这本书，让他们也能避免 bug。
- 遵循本书的风格原则，特别是第 7 章所述的编码风格。由此，能够更好地理解程序，这样就能减少 bug。

- 在具体编写代码之前先做设计。在编写代码的同时进行设计往往会带来“反复返工”的设计，这种设计更难理解，也更容易出错。而且有可能忽略可能的边缘情况和错误条件。
- 利用代码审查。至少应当有两个人查看你写的每一行代码。有时需要采用一种全新的观点才能注意到问题的存在。
- 测试、测试再测试。遵循第 19 章所述的原则。
- 预料到可能的错误条件，并适当地加以处理。特别地，要对内存用光的条件做出规划，并进行处理。这些条件很可能出现，请见第 15 章。
- 最后一条，可能也是最重要的一条，使用智能指针来避免内存泄漏。有关详细内容请见第 13 章、第 15 章和第 25 章。

## 20.4 找出 bug 的方法

程序中应当包含一些特性，以便在出现不可避免的 bug 时能够方便调试。这一节会介绍这些特性，还将提供一些示例实现，可以把这些实现结合到自己的程序中去。

### 20.4.1 错误日志

假想有这样一种情况：你刚刚发布了你的旗舰产品的最新版本，最早的一批用户中，有人报告称这个程序“停止工作了”。你想从用户那里打探到更多信息，最后发现程序是在一个操作中死掉的。用户想不起来他做了什么，也不记得到底有没有错误消息。怎么调试这个问题呢？

在这种情况下，除了能从用户那里得到有限的信息外，还能分析用户计算机上的错误日志。在日志中，你发现来自程序的一个消息称“Error: unable to allocate memory.”（错误：无法分配内存）。查看产生错误消息位置附近的代码，发现有一行中要对一个指针解除引用，但是没有检查该指针是否为 NULL。此时，你就发现了 bug 的根本原因！

错误日志是一个将错误消息写至持久存储的过程，这样在应用终止（甚至机器死机）之后也能得到原来的错误消息。尽管有了上述示例场景，你可能对这个策略还心存疑虑。如果程序遇到错误，从它的行为不就能很明显地看出来吗？如果出问题了，用户难道不会注意到吗？如前例所示，用户报告并不一定准确或完备。另外，许多程序，如操作系统内核和长期运行的后台守护程序（如 UNIX 上的 `inetd` 或 `syslogd`）并不是交互式的，这些程序在机器上运行时无需用户参与，它们与用户通信的惟一途径就是通过错误日志。

因此，程序遇到错误时就应当将错误记入日志。这样一来，如果用户报告了一个 bug，就能分析机器上的日志文件，查看程序在遇到这个 bug 之前是否报告过错误。遗憾的是，错误日志是依赖于平台的，C++ 并没有包含一个标准的日志机制。举例来说，特定于平台的日志机制有 UNIX 中的 `syslog` 工具以及 Windows 中的事件报告 API。可以参考开发平台相关文档。另外还有一些跨平台日志类的开源实现，包括 `log4cpp`（可以从 <http://sourceforge.net> 获得）。

既然已经认识到错误日志是一个可以加到程序中的不错特性，你可能想对代码中的每几行都加上供日志记录的错误消息，这样一来，万一遇到了 bug，就能跟踪所执行的代码路径。这种错误消息就适当地称为“跟踪轨迹”（trace）。不过，不能编写这种跟踪轨迹来建立错误日志，其原因有二。首先，日志写至持久存储是很慢的。即使在异步写日志的系统上，记录那么多的信息也会使程序变慢。其次，也是最重要的，置于跟踪轨迹中的大多数信息并不适合最终用户查看。它可能只会把用户搞糊涂，以至于动不动就打电话寻求服务。也就是说，跟踪是在适当情况下的一个重要调试技术，有关内容请见下一节的介绍。

以下是一些具体的指导原则，在此指出了程序应当把哪些类型的错误记入日志：

- 不可恢复的错误，如无法分配内存，或系统调用意外失败。这些错误通常会直接导致应用提前退

出或内存核心转储 (core dump)。

- 管理员可以采取行动的错误，如内存过少、无法写至磁盘，或者一个网络连接中断。
- 意外错误，如意外的代码路径，或者变量取了未预料到的值。需要注意，代码应当“预想到”用户会输入非法的数据，而且应该适当地加以处理。意外的错误会在程序中表现为一个 bug。
- 安全漏洞，如试图从一个未授权地址建立网络连接，或者试图建立过多网络连接（拒绝服务攻击）。

另外，大多数 API 都允许指定日志级别 (log level) 或错误级别 (error level)。可以将日志级别的非错误条件记入日志，它们没有“错误”那么严重。例如，可能想把应用中显著的状态改变记入日志，或者记录程序的启动和关闭。还可以考虑为用户提供一种办法，可以在运行时调整程序的日志级别，这样他们就能定制能记录多少日志。

## 20.4.2 调试轨迹

在调试复杂的问题时，公开的错误消息通常都没有包含足够的信息。往往需要对出现 bug 之前所取的代码路径或变量值有一个完备的跟踪轨迹。除了基本消息外，有时在调试轨迹中加入以下信息会很有帮助：

- 如果是一个多线程程序，可以加入线程 ID。
- 生成轨迹的函数的函数名。
- 生成此轨迹的代码所在源文件的文件名。

可以通过一个特殊的调试模式，或者通过一个环缓冲区 (ring buffer)，将调试轨迹增加到程序中。这两种方法将在下面详细解释。

### 调试模式

增加调试轨迹的第一个技术是为程序提供一个调试模式。采用调试模式，程序会把轨迹输出写至标准错误输出，或者写至一个文件，还有可能在执行过程中完成一些额外的检查。为程序增加调试模式有许多方式。

#### 编译时调试模式 (Compile-Time Debug Mode)

可以使用预处理器 `#ifdefs` 有选择地将调试代码编译到程序中。这种方法的好处是，调试代码不会编译到“发布”的二进制版本中，因此不会增加它的规模。缺点在于，在客户那里没有办法支持调试来完成测试或发现 bug，而且代码看上去会很杂乱，不好理解。

这一节余下的部分将显示一个例子，这是一个简单的程序，其中提供了编译时调试模式。这个程序并没有做什么有用的工作，它的目的只是为了展示这个技术。

为了生成程序的一个调试版本，在编译时应当已经定义了符号 `DEBUG_MODE`。编译器应当允许你指定在编译时要定义的符号。有关详细内容可以参考你的文档。例如，`g++` 允许在 `compile` 命令上指定 `-Dsymbol`。

注意，这个例子对于 `ofstream` 对象使用了一个全局变量。我们一般不建议使用全局变量，这个例子是少有的几个例外之一。这里之所以可以接受全局变量，原因是调试模式应当不会妨碍余下的程序。如果 `ofstream` 对象不是全局的，就要把它传递给每一个函数，这就需要修改整个程序中的所有函数原型。

```
// CTDebug.cpp
#include <exception>
#include <fstream>
#include <iostream>
using namespace std;
#ifdef DEBUG_MODE
ofstream debugOstr;
```

```

const char* debugFileName = "debugfile.out";
#endif

class ComplicatedClass
{
public:
    ComplicatedClass() {}

    // Class details omitted for brevity
};

class UserCommand
{
public:
    UserCommand() {}

    // Class details not shown for brevity
};

ostream& operator<<(ostream& ostr, const ComplicatedClass& src);
ostream& operator<<(ostream& ostr, const UserCommand& src);
UserCommand getNextCommand(ComplicatedClass* obj);
void processUserCommand(UserCommand& cmd);
void trickyFunction(ComplicatedClass* obj) throw(exception);

int main(int argc, char** argv)
{
#ifdef DEBUG_MODE
    // Open the output stream.
    debugOstr.open(debugFileName);
    if (debugOstr.fail()) {
        cout << "Unable to open debug file!\n";
        return (1);
    }

    // Print the command-line arguments to the trace.
    for (int i = 0; i < argc; i++) {
        debugOstr << argv[i] << " ";
        debugOstr << endl;
    }
#endif

    // Rest of the function not shown
    return (0);
}

ostream& operator<<(ostream& ostr, const ComplicatedClass& src)
{
    ostr << "ComplicatedClass";
    return (ostr);
}

ostream& operator<<(ostream& ostr, const UserCommand& src)
{
    ostr << "UserCommand";
    return (ostr);
}

```

```
UserCommand getNextCommand(ComplicatedClass* obj)
{
    UserCommand cmd;
    return (cmd);
}

void processUserCommand(UserCommand& cmd)
{
    // Details omitted for brevity
}

void trickyFunction(ComplicatedClass* obj) throw(exception)
{
#ifdef DEBUG_MODE
    // If in debug mode, print the values with which this function starts
    debugOstr << "trickyFunction(): given argument: " << *obj << endl;
#endif

    while (true) {
        UserCommand cmd = getNextCommand(obj);

#ifdef DEBUG_MODE
        debugOstr << "trickyFunction(): retrieved cmd " << cmd << endl;
#endif
        try {
            processUserCommand(cmd);
        } catch (exception& e) {
#ifdef DEBUG_MODE
            debugOstr << "trickyFunction(): "
                << "received exception from procesUserCommand(): "
                << e.what() << endl;
#endif
            throw;
        }
    }
}
```

#### 开始时调试模式 (Start-Time Debug Mode)

开始时调试模式是 `#ifdefs` 的一个替代做法，它的实现同样简单。程序的一个命令行参数可以指定是否以调试模式运行。不同于编译时调试模式，这种策略要把调试代码包含在“发布”的二进制版本中，并允许在顾客那里启用调试模式。不过，它还需要用户重启程序，这样才能以调试模式运行，客户通常不喜欢这种做法，而这可能会阻碍你得到有关 bug 的有用信息。

以下开始时调试模式的例子使用了以上的同一个程序（展示编译时调试模式也用了这个程序），这样就能直接地比较出二者的差异。这个版本的程序也使用了全局变量，这一次是对 `ofstream` 使用全局变量，另外存在一个布尔量来指定程序是否处在调试模式，对于这个布尔量也作为一个全局变量。这里之所以可以接受这种选择，是为了避免把额外的调试参数转加到所有的函数原型上。

这个程序中有一个方面需要进一步说明。在 C++ 中没有解析命令行参数的标准库功能。这个程序使用了一个简单的函数 `isDebugSet()` 来检查所有命令行参数中是否存在一个调试标志，不过解析所有命令行参数的函数肯定会更复杂。



```
// STDebug.cpp
#include <exception>
#include <fstream>
#include <iostream>
using namespace std;

ofstream debugOstr;
bool debug = false;

const char* debugFileName = "debugfile.out";
```

```
class ComplicatedClass
{
public:
    ComplicatedClass() {}
    ~ComplicatedClass() {}
};
```

```
class UserCommand
{
public:
    UserCommand() {}
};
```

```
bool isDebugSet(int argc, char** argv);
ostream& operator<<(ostream& ostr, const ComplicatedClass& src);
ostream& operator<<(ostream& ostr, const UserCommand& src);
UserCommand getNextCommand(ComplicatedClass* obj);
void processUserCommand(UserCommand& cmd);
void trickyFunction(ComplicatedClass* obj) throw(exception);
```

```
int main(int argc, char** argv)
{
```

```
    debug = isDebugSet(argc, argv);
    if (debug) {
        // Open the output stream.
        debugOstr.open(debugFileName);
        if (debugOstr.fail()) {
            cout << "Unable to open debug file!\n";
            return (1);
        }

        // Print the command-line arguments.

        for (int i = 0; i < argc; i++) {
            debugOstr << argv[i] << " ";
            debugOstr << endl;
        }

        // Rest of the function not shown
        return (0);
    }
}
```

```
bool isDebugSet(int argc, char** argv)
{
    for (int i = 0; i < argc; i++) {
```

```

        if (strcmp(argv[i], "-d") == 0) {
            return (true);
        }
    }
    return (false);
}

```

```

ostream& operator<<(ostream& ostr, const ComplicatedClass& src)
{
    ostr << "ComplicatedClass";
    return (ostr);
}

```

```

ostream& operator<<(ostream& ostr, const UserCommand& src)
{
    ostr << "UserCommand";
    return (ostr);
}

```

```

UserCommand getNextCommand(ComplicatedClass* obj)
{
    UserCommand cmd;
    return (cmd);
}

```

```

void processUserCommand(UserCommand& cmd)
{
    // Details omitted for brevity
}

```

```

void trickyFunction(ComplicatedClass* obj) throw(exception)
{
    if (debug) {
        // If in debug mode, print the values with which this function starts
        debugOstr << "trickyFunction(): given argument: " << *obj << endl;
    }

    while (true) {
        UserCommand cmd = getNextCommand(obj);
        if (debug) {
            debugOstr << "trickyFunction(): retrieved cmd " << cmd << endl;
        }
        try {
            processUserCommand(cmd);
        } catch (exception& e) {
            if (debug) {
                debugOstr << "trickyFunction(): "
                    << " received exception from procesUserCommand(): "
                    << e.what() << endl;
            }
            throw;
        }
    }
}

```

### 运行时调试模式 (Run-Time Debug Mode)

提供调试模式最灵活的方法就是允许在运行时启用或禁用调试模式。提供这个特性的一种方法是提供一个异步接口，由它实时控制调试模式。在一个 GUI 程序中，这个接口可能采用一个菜单命令的形式。在一个 CLI 程序中，此接口可能是一个异步命令，它会对程序做出一个进程间调用（例如，使用 socket 或远程过程调用）。C++ 对于完成进程间通信或 GUI 没有提供标准方法，所以在此不再显示这个技术的例子。

### 环缓冲区

调试模式对于调试可再生的问题和运行测试很有用。不过，bug 通常都是在程序以非调试模式运行时出现的，而且等你或顾客启用调试模式时，可能已经太晚，得不到 bug 的任何信息了。对这个问题的一个解决方案就是一直启用程序中的轨迹跟踪。通常只需要最近的跟踪轨迹来调试一个程序，所以应当只在程序执行中的某一点保存最近的跟踪轨迹。为此可以精心地使用日志文件循环来实现。

不过，为了避免前面在 20.4.1 节中描述的有关记录跟踪轨迹的问题，最好是程序根本不记录这些轨迹；而把它们保存在内存中。然后，应该提供一种机制，在需要时将所有跟踪轨迹消息转储到标准错误输出或一个日志文件中。一种常用的技术就是使用环缓冲区来存储固定数目的消息，或者在固定大小的内存中存储消息。当缓冲区满时，它会再从缓冲区开始处写消息，覆盖原来的老消息。这个循环可以无限重复。以下小节将提供环缓冲区的一个实现，并说明如何在程序中使用这个环缓冲区实现。

### 环缓冲区接口

```
#include <vector>
#include <string>
#include <fstream>

using std::string;
using std::vector;
using std::ostream;

//
// class RingBuffer
//
// Provides a simple debug buffer. The client specifies the number
// of entries in the constructor and adds messages with the addEntry()
// method. Once the number of entries exceeds the number allowed, new
// entries overwrite the oldest entries in the buffer.
//
// The buffer also provides the option to print entries as they
// are added to the buffer. The client can specify an output stream
// in the constructor, and can reset it with the setOutput() method.
//
// Finally, the buffer supports streaming to an output stream.
//
class RingBuffer
{
public:
    //
    // Constructs a ring buffer with space for numEntries.
    // Entries are written to *ostr as they are queued.
    //
    RingBuffer(int numEntries = kDefaultNumEntries, ostream* ostr = NULL);
```

```

~RingBuffer();

//
// Adds the string to the ring buffer, possibly overwriting the
// oldest string in the buffer (if the buffer is full).
//
void addEntry(const string& entry);

//
// Streams the buffer entries, separated by newlines, to ostr.
//
friend ostream& operator<<(ostream& ostr, const RingBuffer& rb);

//
// Sets the output stream to which entries are streamed as they are added.
// Returns the old output stream.
//
ostream* setOutput(ostream* newOstr);

protected:
    vector<string> mEntries;
    ostream* mOstr;

    int mNumEntries, mNext;
    bool mWrapped;

    static const int kDefaultNumEntries = 500;

private:
    // Prevent assignment and pass-by-value.
    RingBuffer(const RingBuffer& src);
    RingBuffer& operator=(const RingBuffer& rhs);
};

```

### 环缓冲区实现

环缓冲区的这个实现存储了固定数目的字符串。每个字符串都必须复制到环缓冲区中，这需要动态分配内存。这种方法肯定不是最高效的解决方案。还有其他做法，可以为缓冲区提供固定大小（字节数）的内存。不过，这样低级 C 字符串和内存管理就会搅和进来，而这是你着力避免的。除非你在编写一个高性能应用，否则这个实现应当足够了。

这个环缓冲区使用了 STL 的向量来存储字符串项。还可以使用一个标准 C 风格的数组。使用 STL 很简单，只是 RingBuffer 的 operator<< 的实现有些难度，其中会用到一些有意思的迭代器。有关迭代器和复制算法的详细内容请参见第 21 章~第 23 章。

```

#include <algorithm>
#include <iterator>
#include <iostream>
#include "RingBuffer.h"

using namespace std;

const int RingBuffer::kDefaultNumEntries;

//

```

```

// Initialize the vector to hold exactly numEntries. The vector size
// does not need to change during the lifetime of the object.
//
// Initialize the other members.
//
RingBuffer::RingBuffer(int numEntries, ostream* ostr) : mEntries(numEntries),
    mOstr(ostr), mNumEntries(numEntries), mNext(0), mWrapped(false)
{
}

RingBuffer::~RingBuffer()
{
}

//
// The algorithm is pretty simple: add the entry to the next
// free spot, then reset mNext to indicate the free spot after
// that. If mNext reaches the end of the vector, it starts over at 0.
//
// The buffer needs to know if the buffer has wrapped or not so
// that it knows whether to print the entries past mNext in operator<<
//
void RingBuffer::addEntry(const string& entry)
{
    // Add the entry to the next free spot and increment
    // mNext to point to the free spot after that.
    mEntries[mNext++] = entry;

    // Check if we've reached the end of the buffer. If so, we need to wrap.
    if (mNext >= mNumEntries) {
        mNext = 0;
        mWrapped = true;
    }

    // If there is a valid ostream, write this entry to it.
    if (mOstr != NULL) {
        *mOstr << entry << endl;
    }
}

ostream* RingBuffer::setOutput(ostream* newOstr)
{
    ostream* ret = mOstr;
    mOstr = newOstr;
    return (ret);
}

//
// This function uses an ostream_iterator to "copy" entries directly
// from the vector to the output stream.
//
// This function must print the entries in order. If the buffer has wrapped,
// the earliest entry is one past the most recent entry, which is the entry
// indicated by mNext. So first print from entry mNext to the end.
//

```

```
// Then (even if the buffer hasn't wrapped) print from the beginning to mNext - 1.
//
ostream& operator<<(ostream& ostr, const RingBuffer& rb)
{
    if (rb.mWrapped) {
        //
        // If the buffer has wrapped, print the elements from
        // the earliest entry to the end.
        //
        copy (rb.mEntries.begin() + rb.mNext, rb.mEntries.end(),
              ostream_iterator<string>(ostr, "\n"));
    }

    //
    // Now print up to the most recent entry.
    // Go up to begin() + mNext because the range is not inclusive on the
    // right side.
    //
    copy (rb.mEntries.begin(), rb.mEntries.begin() + rb.mNext,
          ostream_iterator<string>(ostr, "\n"));

    return (ostr);
}
```

### 使用环缓冲区

为了使用这个环缓冲区，可以简单地声明一个对象，开始向这个缓冲区增加消息。如果想打印这个缓冲区，只需使用 `operator<<` 将其打印到适当的 `ostream`。以下对较早前的开始时调试模式程序做了修改，以显示环缓冲区的使用。

```
#include "RingBuffer.h"
#include <exception>
#include <fstream>
#include <iostream>
#include <cassert>
```

```
#include <sstream>
using namespace std;
```

```
RingBuffer debugBuf;
```

```
class ComplicatedClass
{
public:
    ComplicatedClass() {}
    ~ComplicatedClass() {}
};
```

```
class UserCommand
{
public:
    UserCommand() {}
};
```

```
ostream& operator<<(ostream& ostr, const ComplicatedClass& src);
ostream& operator<<(ostream& ostr, const UserCommand& src);
UserCommand getNextCommand(ComplicatedClass* obj);
```



```

void processUserCommand(UserCommand& cmd);
void trickyFunction(ComplicatedClass* obj) throw(exception);

int main(int argc, char** argv)
{
    // Print the command-line arguments.
    for (int i = 0; i < argc; i++) {
        debugBuf.addEntry(argv[i]);
    }

    trickyFunction(new ComplicatedClass());

    // Print the current contents of the debug buffer to cout.
    cout << debugBuf;

    return (0);
}

ostream& operator<<(ostream& ostr, const ComplicatedClass& src)
{
    ostr << "ComplicatedClass";
    return (ostr);
}

ostream& operator<<(ostream& ostr, const UserCommand& src)
{
    ostr << "UserCommand";
    return (ostr);
}

UserCommand getNextCommand(ComplicatedClass* obj)
{
    UserCommand cmd;
    return (cmd);
}

void processUserCommand(UserCommand& cmd)
{
    // Details omitted for brevity
}

void trickyFunction(ComplicatedClass* obj) throw(exception)
{
    assert(obj != NULL);

    // Trace log the values with which this function starts.
    ostringstream ostr;
    ostr << "trickyFunction(): given argument: " << *obj;
    debugBuf.addEntry(ostr.str());

    while (true) {
        UserCommand cmd = getNextCommand(obj);

        ostringstream ostr;
        ostr << "trickyFunction(): retrieved cmd " << cmd;
        debugBuf.addEntry(ostr.str());

        try {
            processUserCommand(cmd);
        } catch (exception& e) {

```

```
string msg = "trickyFunction(): received exception from procesUserCommand():";
msg += e.what();
debugBuf.addEntry(msg);
throw;
```

```
    }
    break;
```

```
    }
}
```

需要注意，环缓冲区的这个接口有时要求你在向缓冲区中增加消息项之前先使用 `ostringstream` 或 `string` 连接来构造字符串。

### 显示环缓冲区内容

在内存中存储轨迹调试消息是一个很好的开始，不过为了更有用，还需要一种方法来访问这些跟踪轨迹以便调试。程序应当提供一个“钩子”（hook）来告诉它要打印消息。这个钩子应当与运行时启用调试所用的接口很相似。另外，如果程序遇到一个致命的错误，导致它退出，就应当在退出之前先把环缓冲区打印到标准错误输出或一个日志文件中。

获得这些消息还有一种办法，就是得到程序的内存转储。每个平台会以不同的方式处理内存转储，所以应该参考一本有关的书，或者咨询通晓你所用平台的专家。

### 20.4.3 断言

<cassert> 库中的 `assert` 宏是一个功能很强大的工具。它取一个布尔表达式，如果表达式计算为 `false`，则打印一个错误消息，并终止程序。如果表达式计算为 `true`，就什么也不做。尽管这种行为听上去好像没有多大帮助，但在有些情况下确实非常有用。由此，可以“要求”程序在 bug 出现的准确位置上把 bug 暴露出来。如果在那一点上没有加断言，程序可能会使用这些不正确的值继续运行，而这个 bug 可能直到很久以后才会出现。因此，利用断言，可以较早地检测到 bug，在没有加断言的情况下则往往做不到。

断言（`assert`）的行为依赖于 `NDEBUG` 预处理器符号。如果这个符号未定义，断言就会起作用，否则会被忽略。编译器通常在编译“调试”版本时会定义这个符号。如果想在运行时代码中仍保留断言，必须指定编译器设置，或者编写自己的断言（这个版本的断言不应受 `NDEBUG` 值的影响）。

如果对变量的状态有所估计的话，应当尽可能地在代码中使用断言。例如，如果调用了一个库函数，它要返回一个指针，而且声称绝对不会返回 `NULL`，这样就可以在函数调用之后放上一个断言来确保该指针不为 `NULL`。

要注意，应当尽可能少做假设。例如，如果你在编写一个库函数，不要对基参数是否合法加断言。而是应当检查参数，如果参数非法则返回一个错误码，或者抛出一个异常。只有在别无其他选择的情况下才使用断言。例如，在开始时调试例子中，函数 `trickyFunction()` 取了一个类型为 `ComplicatedClass*` 的参数，不要假设该参数是合法的，更好的做法可能是如下加断言。

```
#include <cassert>
```

```
void trickyFunction(ComplicatedClass* obj) throw(exception)
```

```
{
```

```
    assert(obj != NULL);
```

```
    // Remainder of the function omitted for brevity
```

```
}
```

要小心，不要把必须执行以保证程序正确的代码错误地放在 `assert` 中。例如，下面这行代码就有问题：`assert (myFunctionCall() != NULL)`。如果一个发布版本去掉了代码中的 `assert`，那么对 `myFunctionCall()` 的调用也会一并丢失！

## 20.5 调试技术

调试程序可能非常麻烦。不过，利用一种系统的方法，调试就会容易得多。调试程序的第一步往往应当是再生 bug。取决于是否可以再生此 bug，接下来采用的方法会有所差异。下面的三个小节将解释如何再生 bug，如何调试可再生的 bug，以及如何调试不可再生的 bug。其余的小节将解释有关调试内存错误和调试多线程程序的有关细节。

### 20.5.1 再生 bug

如果你能一致地再生 bug，确定根本原因就会容易得多。所有可再生的 bug 都可以找到根本原因，并得到修正。不可再生的 bug 则很难（甚至不可能）找到根本原因。要再生 bug，第一步是用同样的输入重新运行程序，即要用 bug 第一次出现时所取的输入来运行程序。一定要确保包括从程序启动到出现 bug 这个过程的所有输入。一个常见的错误是力图只完成触发动作就想再生 bug。这个技术可能无法使 bug 再生，因为这个 bug 可能是由整个动作序列导致的。例如，如果请求某个 Web 网页时，Web 浏览器程序由段冲突而终止，这可能是因为这个请求的网址导致了内存冲突。另一方面，也可能是因为程序会把所有请求都存储在一个队列中，这个队列的空间只能容留一百万项，当前的这个请求恰好是第一百万零一项。重启程序，再发送一个请求当然不会导致原来的 bug 再出现。

有时不太可能模拟导致 bug 的整个事件序列。也许有人报告了 bug，但他并不记得自己做了些什么。还有一种可能，也许程序运行的时间太长，以至于无法模拟每一个输入。在这种情况下，就要尽最大努力再生 bug。有时可能要做些猜测，而且这可能很耗费时间，不过此时做出的努力是有回报的，这会在以后节省调试的时间。以下是可以尝试的一些技术：

- 在正确的环境下重复触发动作，并尽可能多地提供与原报告类似的输入。
- 运行能够实现类似功能的自动化测试。再生 bug 正是自动化测试所擅长的。如果 bug 出现前要花 24 个小时，更可取的方法是让这些测试自行运行，而不要亲自花上 24 个小时来再生这个 bug。
- 如果有可用的必要硬件，同时在不同机器上运行稍有不同的测试有时会节省时间。
- 运行提供类似功能的压力测试。如果程序是一个 Web 服务器，因为某个请求终止了，可以尝试同时运行数百万个浏览器做此请求。

一致地再生 bug 之后，应当确定再生此 bug 的最简单、最高效的测试用例是什么。这样可以更简单地找到问题的根本原因，而且更易于验证修正。

### 20.5.2 调试可再生 bug

如果能一致而且高效地再生一个 bug，下面就要找出代码中哪个问题导致了这个 bug。此时的目标是要准确地找到导致问题的那几行代码。可以使用两种不同的策略。

1. `cout` 调试。通过为程序增加足够多的调试消息，并在再生 bug 时查看其输出，就能准确地指出出现 bug 的那几行代码了。如果手边有可用的调试工具，通常不建议采用这种方法，因为这需要修改程序，而且可能很耗费时间。不过，如果如前所述已经在程序中加入了调试消息，那么只需在再生 bug 时以调试模式运行程序，就可以找出 bug 的根本原因。这个技术可能比依靠一个调试工具更快一些。

2. 使用一个调试工具。希望你对调试工具已经很熟悉了，利用调试工具可以逐步跟踪程序的执行，并查看各个点上内存的状态和变量的值。如果你还没有使用过调试工具，应当尽快地学习如何来使用。要找出 bug，调试工具通常是不可少的工具。如果能访问源代码，就可以使用一个符号调试工具（symbolic debugger），这种调试工具会利用变量名、类名和代码中的其他符号。为了使用一个符号调试工具，编译程序时必须包括调试信息。否则，符号信息会从程序可执行文件中去除，这样调试工具就得不到这些符号信息了。本章最后的调试例子展示了这两种方法。

### 20.5.3 调试不可再生 bug

如果是不可再生的 bug，修正这种 bug 要比查找可再生 bug 的根本原因困难多了。通常你只能得到很少的信息，而且必须做大量的猜测。不过，还是有一些策略能够对你有所帮助。

1. 尝试将一个不可再生的 bug 转变成为一个可再生的 bug。通过合理的猜测，通常可以大致明确 bug 出现在哪里。花些时间来再生这个 bug 是值得的。一旦有了可再生的 bug，就可以使用前面所述的技术得出根本原因。

2. 分析错误日志。在理想情况下，如前所述，已经让程序生成了错误日志。应当在这个信息中筛选有用的信息，因为 bug 出现之前记录的错误很可能会对这个 bug 本身产生作用。如果很幸运（或者写的程序很好），程序可能会记录下当前 bug 的准确原因。

3. 得到并分析跟踪轨迹。理想情况下，已经如前所述通过一个环缓冲区来提供轨迹输出。bug 出现时，你可能已经得到了一个轨迹副本。这些跟踪轨迹可以指示出代码中 bug 的准确位置。

4. 分析一个内存转储（memory dump）文件（如果有这样一个文件的话）。有些平台会为异常终止的应用生成内存转储文件。在 UNIX 上，这些内存转储文件称为核心文件（core file）（译者注：即前面提到的核心转储）。每个平台都为分析这些内存转储文件提供了分析工具。即便没有符号调试信息，通常也可以从这些文件得到大量的信息。例如，通常可以生成应用终止前的栈轨迹，因为诸如函数和方法名等全局符号往往可以从二进制版本中得到。如果你对所用平台的汇编语言很熟悉，可以对机器码反汇编，得到汇编代码。另外，可以查看内存的内容，不过，倘若没有符号，这是无类型（untyped）而且无名（unnamed）的。

5. 查看代码。遗憾的是，这通常是确定不可再生 bug 原因的惟一策略。令人诧异的是，这往往都能奏效。在检查代码时，由于刚刚出现了 bug，从这个角度看代码，即使是自己写的代码，通常也可以发现原来忽视的错误。我们并不建议你花上几个小时盯着代码不放，不过跟踪代码路径确实可以让你找到问题所在。

6. 使用一个内存查看工具，如以下“调试内存问题”一节中介绍的工具。这些工具会提醒你存在一些内存错误，尽管这些错误并不一定会导致程序表现异常，但是这可能就是当前 bug 的原因。

7. 提交或更新一个 bug 报告。即使不能立即找出 bug 的根本原因，做出相关报告也能很好地记录下你所做的努力，以备再次遇到同样的问题。有关 bug 跟踪系统的详细内容请参见第 19 章。

一旦找出了不可再生 bug 的根本原因，就应当创建一个可再生的测试用例，并把它移到“可再生 bug”一类中。真正修正 bug 之前，能够再生 bug 是相当重要的。否则，怎么来测试到底修正了没有呢？调试不可再生 bug 时有一个常见的错误，就是修正了代码中错误的问题（译者注：本来要修正一个问题，但是却修正了另一个问题）。由于无法再生 bug，就无法知道是否真的将问题修正。倘若如此，bug 有可能在一个月以后再次出现，对此你也不必感到奇怪。

### 20.5.4 调试内存问题

大多数灾难性 bug（如应用终止）都可能是因为内存错误造成的。许多非灾难性 bug 也要归因于内存

错误。有些内存 bug 很明显，如果程序对一个 NULL 指针解除引用，它就会立即终止。不过，其他一些则更阴险一些。如果在 C++ 中对数组的写操作越界了，程序可能不会就在此时直接崩溃。不过，如果数组在栈上，可能会写到一个不同的变量或数组中，而这在以后才会暴露出来。或者，如果数组在堆上，可能会导致堆中内存破坏，这样以后当你试图动态分配或释放更多内存时，就会出现错误。第 13 章从编写代码时要避免哪些内存错误的角度介绍了一些常见的内存错误。这一节将从另一个角度来讨论内存错误，即如何找出出现 bug 的代码中的问题。在阅读这一节之前，应当先熟悉第 13 章讨论的内容。

### 内存错误分类

要调试内存问题，应当熟悉可能会出现哪些类型的错误。这一节将介绍内存错误的主要分类。每个内存错误都包括一个小的代码例子，展示了相关错误，以及可能观察到的症状（symptom）。注意，症状和 bug 本身并不是同一个东西。症状是 bug 所导致的可观察到的行为。

### 内存释放错误

表 20-1 总结了涉及释放内存的 5 个主要错误。

表 20-1

错误类型	症 状	示 例
内存泄漏	进程随时间变大 进程随时间运行得越来越慢 最终，由于内存不足，命令和系统调用失败	<pre>void memoryLeak() {     int* ip = new int [1000];     return; // Bug! Not freeing ip. }</pre>
使用不匹配的分配和释放命令	并不总会立即导致程序崩溃 在某些平台上会导致内存破坏，可能在以后表现为程序崩溃（段冲突）	<pre>void mismatchedFree() {     int* ip1 = (int*) malloc (sizeof (int));     int* ip2 = new int;     int* ip3 = new int [1000];     delete ip1; // BUG! Should use free     delete [] ip2; // BUG! Should use delete     free (ip3); // BUG! Should use delete [] }</pre>
多次释放内存	如果在两个撤销调用之间该位置的内存已经另做分配，可能导致一个程序崩溃（段冲突）	<pre>void doubleFree() {     int* ip1 = new int [1000];     delete [] ip1;     int* ip2 = new int [1000];     delete [] ip1; // BUG! freeing ip1 twice }</pre>

(续)

错误类型	症 状	示 例
释放未分配的内存	通常会导致一个程序崩溃 (段冲突或总线错误)	<pre>void freeUnallocated() {     int* ip1 =     reinterpret_cast&lt; int* &gt; (10000);     // BUG! ip1 is not a valid pointer.     delete ip1; }</pre>
释放栈内存	理论上讲, 这是释放未分配 内存的一个特例。通常会导致 一个程序崩溃	<pre>void freeStack() {     int x;     int* ip = &amp;x;     delete ip; // BUG! Freeing stack memory }</pre>

可以看到, 有些内存释放错误并不会立即导致程序终止。这些 bug 很阴险, 会导致程序在以后的运行中出问题。

内存访问错误

第二类内存错误涉及具体的内存读写, 如表 20-2 所示。

表 20-2

错误类型	症 状	示 例
访问非法内存	总是导致程序立即崩溃	<pre>void accessInvalid() {     int* ip1 =     reinterpret_cast&lt; int* &gt; (10000);     // BUG! ip1 is not a valid pointer.     * ip1 = 5; }</pre>
访问已经释放的内存	并不总是导致程序崩溃。如果内存已经另做分配, 可能会意外地出现“奇怪”的值	<pre>void accessFreed() {     int* ip1 = new int;     delete ip1;     int* ip2 = new int;     // BUG! The memory pointed to by ip1     // has been freed.     * ip1 = 5; }</pre>



(续)

错误类型	症 状	示 例
访问另一个分配中的内存	不会导致程序崩溃 可能导致意外地出现“奇怪”的值	<pre> void accessElsewhere() {     int x, y [10], z;     x = 0;     z = 0;     // BUG! element 10 is past the     // end of the array.     for (int i = 0; i &lt;= 10; i++) {         y[i] = 10;     } } </pre>
读取未初始化的内存	不会导致程序崩溃，除非把未初始化的值用作一个指针，并对其解除引用（如此例所示）。即便如此，也不一定导致程序崩溃	<pre> void readUninitialized() {     int* ip;     // BUG! ip is uninitialized.     cout &lt;&lt; *ip &lt;&lt; endl; } </pre>

与内存释放错误相比，内存访问错误更容易导致程序崩溃。不过，也不总会这样。它们也可能导致程序中出现小的非灾难性 bug。

#### 调试内存错误的提示

每次运行程序时，与内存相关的 bug 通常出现在代码中稍有不同的地方。堆内存破坏就是如此。堆内存破坏就像是一个定时炸弹，在你想要在堆上分配、释放或使用内存时，这个炸弹随时准备爆炸。因此，如果看到一个可再生的 bug，但是它在稍有不同的位置出现，就可以怀疑是内存破坏。例如，程序可能出现一次段冲突，随后的一次则出现一个总线错误。如果怀疑存在内存 bug，最好的选择就是使用 C++ 的一个内存检查工具。调试工具通常提供了一些选项，可以在运行程序同时检查内存错误。另外，还有一些非常棒的第三方工具，如 Rational Software 的 purify（现在归 IBM 所有）或 Linux 的 valgrind（见第 13 章的讨论）。这些调试工具都会提供自己的内存分配和释放例程，从而检查出对动态内存的误用，如释放了未分配的内存，对未分配的内存解除引用，或者对数组的写越界。

如果手边没有可用的内存检查工具，而且一般的调试策略无济于事，可能就需要靠代码审查了。一旦缩小了包含 bug 的代码范围，就可以查看是否存在以下问题：

#### 与对象和类相关的错误

- 验证类（涉及动态分配内存）是否有析构函数，而且该析构函数是否准确地释放了对象中分配的内存：不多也不少。
- 确保类利用复制构造函数和赋值操作符正确地处理了复制和赋值，有关内容见第 9 章。
- 检查可疑的强制转换。如果把一个指针从一个类型强制转换为另一个类型的对象，一定要保证这是合法的。

#### 一般性内存错误

- 确保对 new 的每一个调用都与一个（仅一个）delete 调用匹配。类似地，每个 malloc、alloc 或 calloc 调用应当与一个 free 调用匹配。对 new [] 的各个调用应当与一个 delete [] 调用匹配。尽

管重复的 free 调用一般没有什么害处,但是如果在第一个 free 之后,这部分内存存在另一次内存分配中又被分配出去,那么第二次调用 free 时,就可能出现问题。

- 检查缓冲区溢出。在迭代处理一个数组,或者对一个 C 风格的字符串进行读写时,要确保没有越界访问内存。
- 检查是否对非法指针解除引用。

### 20.5.5 调试多线程程序

与 Java 中不同, C++ 语言没有为线程和线程间的同步提供相应机制。不过,多线程的 C++ 程序很常见,所以考虑调试一个多线程程序时涉及的特殊问题很重要。多线程程序中的 bug 通常是因为操作系统调度中计时变化导致的,而且很难再生。

因此,调试多线程程序要采用一组特殊的技术。

1. 使用 cout 调试。调试多线程程序时, cout 调试往往比使用一个调试工具更有效。大多数调试工具都没有很好地处理多线程的执行,或者至少没有为调试多线程程序提供方便。如果不知道给定时刻哪个线程会运行,跟踪程序将会很困难。要在程序中的临界段(critical section)前后增加调试语句,还要在获取锁之前和释放锁之后增加调试语句。通过查看此输出,就能检测出死锁和竞争条件,因为能看到两个线程可能同时位于一个临界段中,或者一个线程因等待一个锁而阻塞。

2. 插入强制的睡眠和上下文切换。如果再生问题麻烦不断,或者找不出根本原因,但是又想进行验证,就可以让线程睡眠按指定的时间,强制发生某些线程调度行为。尽管在 C++ 中对于如何让线程睡眠没有标准方法,不过大多数平台都提供了一个名为 sleep() 的调用。如果在释放一个锁前睡眠几秒,或者恰在通知一个条件变量或访问共享数据之前睡眠,这些都可以暴露出竞争条件,否则可能无法检测到这些竞争条件。

### 20.5.6 调试示例:文章引用

这一节将提供一个有 bug 的程序,以此显示可以采取哪些步骤来调试 bug 并修正问题。

假设你参加一个团队,共同编写一个 Web 页面,允许用户搜索引用了某篇论文的研究文章。如果有作者想了解有哪些与自己类似的工作,这个服务就很有用。如果他们找到一篇介绍相关工作的论文,就可以查找引用这篇论文的所有文章,来寻找其他的相关工作。

在这个项目中,你要负责编写将原始引用数据从文本文件中读入的代码。为简单起见,假设每篇论文的引用信息都可在自己的文件中找到。另外,假设每个文件的第一行包含文章的作者、标题和出版信息,第二行总是为空,后面的行则包含对此文章的引用(每一行是一个引用)。以下是一个示例文件,它针对的是计算机科学领域最著名的文章之一(Alan Turing 的一篇经典文章)。

```
Alan Turing, "On Computable Numbers with an Application to the Entscheidungsproblem", \
Proceedings of the London Mathematical Society, Series 2, Vol.42 (1936 - 37) pages \
230 to 265.
Godel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter \
Systeme, I", Monatshefte Math. Phys., 38 (1931), 173-198.
Alonzo Church, "An unsolvable problem of elementary number theory", American J of \
Math., 58(1936), 345-363.
Alonzo Church, "A note on the Entscheidungsproblem", J. of Symbolic logic, 1 (1930), \
40-41.
Cf. Hobson, "Theory of functions of a real variable (2nd ed., 1921)", 87, 88. \
Proc. London Math. Soc (2) 42 (1936-7), 230-265.
```

注意\ 字符是连续字符，以确保计算机在处理时会把多行作为一行来处理。

#### 有 bug 的 ArticleCitations 类实现

你决定编写一个 ArticleCitations 类来读取文件并存储此信息。这个类把第一行的文章信息读入一个字符串，并把引用信息读入一个字符串数组。请注意，这个设计决定不一定是最好的选择。不过，我们的目的是展示有 bug 的应用，所以这样设计很不错，可以充分反映问题。类定义如下所示：

```
#include <string>
using std::string;

class ArticleCitations
{
public:
    ArticleCitations(const string& fileName);
    ~ArticleCitations();
    ArticleCitations(const ArticleCitations& src);
    ArticleCitations& operator=(const ArticleCitations& rhs);
    string getArticle() const { return mArticle; }
    int getNumCitations() const { return mNumCitations; }
    string getCitation(int i) const { return mCitations[i]; }

protected:
    void readFile(const string& fileName);

    string mArticle;
    string* mCitations;
    int mNumCitations;
};
```

方法实现如下。这个程序是有 bug 的，不要把它当作模型照搬使用。

```
#include "ArticleCitations.h"
#include <iostream>
#include <fstream>
#include <string>
#include <stdexcept>
using namespace std;

ArticleCitations::ArticleCitations(const string& fileName)
{
    // All we have to do is read the file.
    readFile(fileName);
}

ArticleCitations::ArticleCitations(const ArticleCitations& src)
{
    // Copy the article name, author, etc.
    mArticle = src.mArticle;
    // Copy the number of citations.
    mNumCitations = src.mNumCitations;
    // Allocate an array of the correct size.
    mCitations = new string[mNumCitations];
    // Copy each element of the array.
    for (int i = 0; i < mNumCitations; i++) {
        mCitations[i] = src.mCitations[i];
    }
}
```

```

    }
}

ArticleCitations& ArticleCitations::operator=(const ArticleCitations& rhs)
{
    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }
    // Free the old memory.
    delete [] mCitations;
    // Copy the article name, author, etc.
    mArticle = rhs.mArticle;
    // Copy the number of citations.
    mNumCitations = rhs.mNumCitations;
    // Allocate a new array of the correct size.
    mCitations = new string[mNumCitations];
    // Copy each citation.
    for (int i = 0; i < mNumCitations; i++) {
        mCitations[i] = rhs.mCitations[i];
    }
    return (*this);
}

ArticleCitations::~ArticleCitations()
{
    delete[] mCitations;
}

void ArticleCitations::readFile(const string& fileName)
{
    // Open the file and check for failure.
    ifstream istr(fileName.c_str());
    if (istr.fail()) {
        throw invalid_argument("Unable to open file\n");
    }
    // Read the article author, title, etc. line.
    getline(istr, mArticle);

    // Skip the white space before the citations start.
    istr >> ws;

    int count = 0;
    // Save the current position so we can return to it.
    int citationsStart = istr.tellg();
    // First count the number of citations.
    while (!istr.eof()) {
        string temp;
        getline(istr, temp);
        // Skip white space before the next entry.
        istr >> ws;
        count++;
    }

    if (count != 0) {
        // Allocate an array of strings to store the citations.
        mCitations = new string[count];
    }
}

```

```

        mNumCitations = count;
        // Seek back to the start of the citations.
        istr.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < mNumCitations; count++) {
            string temp;
            getline(istr, temp);
            mCitations[count] = temp;
        }
    }
}

```

### 测试 ArticleCitations 类

按照第 19 章的建议，你决定在继续下一步工作前对 ArticleCitations 类做单元测试，不过在这个例子中为简单起见，这个单元测试没有使用测试框架。以下程序要求用户提供一个文件名，用该文件名构造一个 ArticleCitations 类，并按值把对象传递给 processCitations() 函数，这个函数会使用对象的公共存取方法把信息打印出来。

```

#include "ArticleCitations.h"
#include <iostream>
using namespace std;

void processCitations(ArticleCitations cit);

int main(int argc, char** argv)
{
    string fileName;

    while (true) {
        cout << "Enter a file name (\\"STOP\\" to stop): ";
        cin >> fileName;

        if (fileName == "STOP") {
            break;
        }
        // Test constructor
        ArticleCitations cit(fileName);
        processCitations(cit);
    }
    return (0);
}

void processCitations(ArticleCitations cit)
{
    cout << cit.getArticle() << endl;
    int num = cit.getNumCitations();
    for (int i = 0; i < num; i++) {
        cout << cit.getCitation(i) << endl;
    }
}

```

### cout 调试

你决定针对 Alan Turing 例子（保存在一个名为 paper1.txt 的文件中）来测试这个程序。输出如下：

```
Enter a file name (*STOP* to stop): paper1.txt
Alan Turing. "On Computable Numbers with an Application to the
Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2,
Vol.42 (1936 - 37) pages 230 to 265.
```

```
Enter a file name (*STOP* to stop): STOP
```

好像不太对！应该打印出 5 个引用信息，而不应该是 5 个空行。针对这个 bug，你想采用 cout 调试。在这个例子中，可以先查看从文件读取引用情况的函数。如果这个函数有问题，显然对象就不会有引用。可以把 readFile() 修改如下：

```
void ArticleCitations::readFile(const string& fileName)
{
    // Open the file and check for failure.
    ifstream istr(fileName.c_str());
    if (istr.fail()) {
        throw invalid_argument("Unable to open file\n");
    }
    // Read the article author, title, etc. line.
    getline(istr, mArticle);

    // Skip the white space before the citations start.
    istr >> ws;

    int count = 0;
    // Save the current position so we can return to it.
    int citationsStart = istr.tellg();
    // First count the number of citations.
    cout << "readFile(): counting number of citations\n";
    while (!istr.eof()) {
        string temp;
        getline(istr, temp);
        // Skip white space before the next entry.
        istr >> ws;
        cout << "Citation " << count << ": " << temp << endl;
        count++;
    }

    cout << "Found " << count << " citations\n" << endl;
    cout << "readFile(): reading citations\n";
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        mCitations = new string[count];
        mNumCitations = count;
        // Seek back to the start of the citations.
        istr.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < mNumCitations; count++) {
            string temp;
            getline(istr, temp);
            cout << temp << endl;
            mCitations[count] = temp;
        }
    }
}
```



再对这个程序运行同样的测试，可以得到以下输出：

```
Enter a file name ("STOP" to stop): paper1.txt
readFile(): counting number of citations
Citation 0: Godel, "Uber formal unentscheidbare Satze der Principia Mathernatica
und verwant der Systeme, I", Monatshefte Math. Phys., 38 (1931). 173-198.

Citation 1: Alonzo Church. "An unsolvable problem of elementary number theory",
American J of Math., 58(1936), 345 363.
Citation 2: Alonzo Church. "A note on the Entscheidungsproblem", J. of Symbolic
logic, 1 (1930), 40 41.
Citation 3: Cf. Hobson, "Theory of functions of a real variable (2nd ed., 1921)",
87, 88.
Citation 4: Proc. London Math. Soc (2) 42 (1936 7), 230 265.
Found 5 citations

readFile(): reading citations
```

Alan Turing, "On Computable Numbers with an Application to the  
Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2,  
Vol.42 (1936 - 37) pages 230 to 265.

Enter a file name ("STOP" to stop):

从输出可以看到，程序第一次从文件读取引用时，为了统计引用数，会正确地读取。不过，第二次的读取则不正确。为什么呢？要想深入分析这个问题，一种做法是再增加一些调试代码，检查每次尝试读取一个引用后文件流的状态：

```
void printStreamState(const istream& istr)
{
    if (istr.good()) {
        cout << "stream state is good\n";
    }
    if (istr.bad()) {
        cout << "stream state is bad\n";
    }
    if (istr.fail()) {
        cout << "stream state is fail\n";
    }
    if (istr.eof()) {
        cout << "stream state is eof\n";
    }
}

void ArticleCitations::readFile(const string& fileName)
{
    // Open the file and check for failure.
```

```

ifstream istr(fileName.c_str());
if (istr.fail()) {
    throw invalid_argument("Unable to open file\n");
}
// Read the article author, title, etc. line.
getline(istr, mArticle);

// Skip the white space before the citations start.
istr >> ws;

int count = 0;
// Save the current position so we can return to it.
int citationsStart = istr.tellg();
// First count the number of citations.
cout << "readFile(): counting number of citations\n";
while (!istr.eof()) {

    string temp;
    getline(istr, temp);
    // Skip white space before the next entry.
    istr >> ws;
    printStreamState(istr);
    cout << "Citation " << count << ": " << temp << endl;
    count++;
}

cout << "Found " << count << " citations\n" << endl;
cout << "readFile(): reading citations\n";
if (count != 0) {
    // Allocate an array of strings to store the citations.
    mCitations = new string[count];
    mNumCitations = count;
    // Seek back to the start of the citations.
    istr.seekg(citationsStart);
    // Read each citation and store it in the new array.
    for (count = 0; count < mNumCitations; count++) {
        string temp;
        getline(istr, temp);
        printStreamState(istr);
        cout << temp << endl;
        mCitations[count] = temp;
    }
}
}

```

这一次运行程序时，会发现一些有意思的信息：

```

Enter a file name ("STOP" to stop): paper1.txt
readFile(): counting number of citations
stream state is good
Citation 0: Godel, "Über formal unentscheidbare Satze der Principia Mathernatica
und verwant der Systeme, I", Monatshefte Math. Phys., 38 (1931). 173-198.
stream state is good
Citation 1: Alonzo Church. "An unsolvable problem of elementary number theory".
American J of Math., 58(1936), 345 363.
stream state is good

```

Citation 2: Alonzo Church. "A note on the Entscheidungsproblem", J. of Symbolic logic, 1 (1930), 40 41.

stream state is good

Citation 3: Cf. Hobson, "Theory of functions of a real variable (2nd ed., 1921)", 87, 88.

stream state is eof

Citation 4: Proc. London Math. Soc (2) 42 (1936 7), 230 265.

Found 5 citations

readFile(): reading citations

stream state is fail

stream state is eof

stream state is fail

stream state is eof

stream state is fail

stream state is eof

stream state is fail

stream state is eof

stream state is fail

stream state is eof

Alan Turing, "On Computable Numbers with an Application to the Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2, Vol.42 (1936 - 37) pages 230 to 265.

Enter a file name ("STOP" to stop):

看上去，在第一次读取引用时，读到最后一个引用之前流状态都是好的。读完最后一个引用（第 5 个引用，即 Citation 4）时流状态为 eof，因为已经到达文件最后。这也是预料之中的事情。但是没有想到的是，第二次所有读取引用的尝试后流状态都同时为 fail 和 eof。猛地看上去，这是没道理的，使用 seekg() 的代码在第二次读引用之前会向前定位到引用开始处，所以不应该还在文件最后位置。不过，还记得第 13 章曾谈到，流会维护错误和 eof 状态，直到你显式将其清除。seekg() 并没有自动地清除 eof 状态。如果处在一个错误或 eof 状态，流就无法正确地读取数据，这也说明了第二次尝试读取引用后为什么流状态为 fail。再仔细地分析一下你的方法，可以看到它在到达文件最后时没有对 istream 调用 clear()。如果修改这个方法，增加一个 clear() 调用，它会正确地读取引用。

以下是正确的 readFile() 方法，其中没有加调试 cout 语句：

```
void ArticleCitations::readFile(const string& fileName)
{
    // CODE OMITTED FOR BREVITY

    if (count != 0) {
        // Allocate an array of strings to store the citations.
        mCitations = new string[count];
        mNumCitations = count;
    }
}
```

```

// Clear the previous eof.
istr.clear();
// Seek back to the start of the citations.
istr.seekg(citationsStart);
// Read each citation and store it in the new array.
for (count = 0; count < mNumCitations; count++) {
    string temp;

    getline(istr, temp);
    mCitations[count] = temp;
}
}
}

```

### 使用调试工具

以下例子使用了 Linux 操作系统上的 gdb 调试工具。

既然 ArticleCitations 类能正确地处理一个引用文件，你决定更进一步，测试一些特殊情况，先从没有引用的文件开始。这个文件如下所示，它存储在一个 paper2.txt 文件中：

Author with no citations

在对这个文件运行程序时，会得到以下结果：

```

Enter a file name ("STOP" to stop): paper1.txt
Alan Turing. "On Computable Numbers with an Application to the
Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2,
Vol.42 (1936 - 37) pages 230 to 265.
Godel, "Uber formal unentscheidbare Satze der Principia Mathernatica und verwant
der Systeme, I", Monatshefte Math. Phys., 38 (1931). 173-198.
Alonzo Church. "An unsolvable problem of elementary number theory", American J of
Math., 58(1936), 345 363.
Alonzo Church. "A note on the Entscheidungsproblem", J. of Symbolic logic, 1
(1930), 40 41.
Cf. Hobson, "Theory of functions of a real variable (2nd ed., 1921)", 87, 88.
Proc. London Math. Soc (2) 42 (1936 7), 230 265.
Enter a file name ("STOP" to stop): paper2.txt
Author with no citations
Segmentation fault

```

肯定是出了某种内存错误。这一次你决定拿调试工具试试看。Gnu DeBugger (gdb) 在 UNIX 平台上都可以得到，而且表现很不错。首先，必须在编译程序时带调试信息（对于 g++ 为 -g）。然后，可以在 gdb 下启动程序。以下是使用这个调试工具寻找问题根本原因的记录示例：

```

>gdb buggyprogram
GNU gdb Red Hat Linux (5.2-2)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "ia64-redhat-linux"...
(gdb) run
Starting program: buggyprogram

```

```
Enter a file name ("STOP" to stop): paper1.txt
Alan Turing, "On Computable Numbers with an Application to the
Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2,
Vol.42 (1936 - 37) pages 230 to 265.
Godel, "Uber formal unentscheidbare Satze der Principia Mathernatica und verwant
der Systeme, I", Monatshefte Math. Phys., 38 (1931). 173-198.
Alonzo Church, "An unsolvable problem of elementary number theory", American J of
Math., 58(1936), 345 363.
Alonzo Church, "A note on the Entscheidungsproblem", J. of Symbolic logic, 1
(1930), 40 41.
Cf. Hobson, "Theory of functions of a real variable (2nd ed., 1921)", 87, 88.
Proc. London Math. Soc (2) 42 (1936 7), 230 265.
Enter a file name {"STOP" to stop}: paper2.txt
Author with no citations
```

```
Program received signal SIGSEGV, Segmentation fault.
__libc_free (mem=0x6000000000010320) at malloc.c:3143
3143 malloc.c: No such file or directory.
    in malloc.c
Current language: auto; currently c
```

SEGV (段冲突) 出现时, 调试工具允许你查看当时程序的状态。bt 命令会显示当前的栈轨迹。可以用 up 和 down 移到栈中上面和下面的函数调用。

```
(gdb) bt
#0 __libc_free (mem=0x6000000000010320) at malloc.c:3143
#1 0x2000000000089010 in __builtin_delete ()
    from /usr/lib/libstdc++-libc6.2-2.so.3
#2 0x2000000000089050 in __builtin_vec_delete ()
    from /usr/lib/libstdc++-libc6.2-2.so.3
#3 0x400000000000a820 in ArticleCitations::~ArticleCitations (
    this=0x80000fffffb920, __in_chrg=2) at ArticleCitations.cpp:51
#4 0x4000000000004f40 in main (argc=1, argv=0x80000fffffb968)
    at BuggyProgram.cpp:20
```

这个栈轨迹中让人感兴趣的一项是 delete 调用了 free()。new 和 delete 以 malloc() 和 free() 的形式来实现是相当常见的。更重要的是, 从这个栈轨迹可以看到, ArticleCitations 析构函数中可能存在某种问题。list 命令可以显示当前栈帧中的代码:

```
(gdb) up 3
#3 0x400000000000a820 in ArticleCitations::~ArticleCitations (
    this=0x80000fffffb920, __in_chrg=2) at ArticleCitations.cpp:51
51 delete [] mCitations;
Current language: auto; currently c++
(gdb) list
46 return (*this);
47 }
48
49 ArticleCitations::~ArticleCitations()
50 {
51 delete [] mCitations;
52 }
53
54 void ArticleCitations::readFile(const string& fileName)
{
```

析构函数中只做了一件事，就是一个 delete [] 调用。在 gdb 中，可以用 print 打印出当前作用域内可以得到的值。为了寻找问题的根本原因，可以打印对象的一些成员变量看看。应该记得 C++ 中 string 类型实际上是一个 typedef，这是对 basic\_string 模板的 char 实例化。

```
(gdb) print mCitations
$3 = (
    basic_string<char, string_char_traits<char>, __default_alloc_template<true, 0> >
*) 0x60000000000010338
```

mCitations 看上去像是一个合法的指针（不过，当然这很难讲）。

```
(gdb) print mNumCitations
$2 = 5
```

找到问题了。这篇文章没有任何引用。为什么 mNumCitations 会置为 5 呢？再来看 readFile() 中针对没有引用的情况的相应代码。在这种情况下，看来它从来也没有初始化 mNumCitations 和 mCitations，代码会取当时恰好在内存中这些位置上的垃圾。在这里，原来的 ArticleCitations 对象的 mNumCitations 值就为 5。第二个 ArticleCitations 对象肯定是放在内存中同一个位置了，所以也会得到同一个值。不过，随机分配的指针值肯定不是一个可以删除的合法指针！无论在文件中是否找到引用，都应当初始化 mCitations 和 mNumCitations。以下是修正后的代码：

```
void ArticleCitations::readFile(const string& fileName)
{
    // CODE OMITTED FOR BREVITY

    mCitations = NULL;
    mNumCitations = 0;
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        mCitations = new string[count];
        mNumCitations = count;
        // Clear the previous eof.
        istr.clear();
        // Seek back to the start of the citations.
        istr.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < mNumCitations; count++) {
            string temp;
            getline(istr, temp);
            mCitations[count] = temp;
        }
    }
}
```

从这个例子可以看到，内存错误并不一定立即暴露出来。通常需要一个调试工具和某种持久存储才能把这种问题找出来。

如果想在另一个平台上得到同样的调试记录，可能发现，由于内存错误的无常变化，可能会与此例所示的情况不同，程序会在其他位置上崩溃。



### 20.5.7 从 ArticleCitations 示例学到的教训

你可能觉得这个例子太小了，不足以代表真正的调试。尽管这段有 bug 的代码确实不长，但你写的许多类也不会比这大多少，即使是在大型项目中也是如此。因此，这个例子更加确证了第 19 章关于单元测试重要性的介绍。假设在把这个例子与项目中余下的内容集成在一起之前，没有充分地对这个例子进行测试，如果这些 bug 以后出现，你和其他工程人员可能要花更多的时间来缩小问题的范围，之后才能如上所示进行调试。另外，这个例子中展示的技术能够适用于所有调试，而不论规模大小。

## 20.6 小结

本章最重要的概念就是调试基本法则：编写代码时要避免 bug，但是要为代码中可能出现的 bug 做出规划。在编程中 bug 总会出现，这是一个事实。如果已经在程序中做好了充分的准备，提供了错误日志、环缓冲区中的调试轨迹，并加了断言，那么具体的调试将会容易得多。

除了这些技术，本章还提供了调试 bug 的一些特定方法。在具体调试时最重要的规则就是让问题再生。然后，可以使用 `cout` 调试或一个符号调试工具跟踪根本原因。内存错误带来的难度更大，而且 C++ 代码中的大多数 bug 都是因为内存错误导致的。这一章介绍了各类内存 bug 及其症状，还通过一个程序给出了调试错误的一些例子。

## 第 21 章 深入 STL：容器和迭代器

许多声称了解 C++ 的程序员其实从未听说过标准模板库。作为一个专业的 C++ 程序员，要充分熟悉标准模板库的强大功能。如果在程序中合理地结合了 STL 容器和算法，而不是从头到尾都自行编写和调试自己的版本，就能节省大量的时间和精力。既然已经学习了第 1 章到第 20 章的内容，你应该已经是一个专业的 C++ 设计人员、编码人员、测试人员和调试人员。现在该掌握 STL 了。

第 4 章谈到过 STL，介绍了 STL 的基本原理，并对各种容器和算法提供了一个概述。你应该对第 4 章的这一节很熟悉了，另外第 2 部分和第 3 部分中大多数章节的内容（特别是第 11 章和第 16 章）也应该不再陌生。

本章将启程开始我们的 STL 之旅，这次“旅行”共包括 3 个部分，这里先介绍 STL 容器，具体内容包括：

- 容器概述。介绍元素的需求，一般的错误处理和迭代器。
- 顺序容器：vector、deque 和 list。
- 容器适配器：queue、priority\_queue 和 stack。
- 关联容器：pair 工具类、map、multimap、set 和 multiset。
- 其他容器：数组、string、流和 bitset。

第 22 章将继续介绍 STL，其中会说明并展示可用于容器元素的一些通用算法例子。第 22 章还会介绍 STL 中预定义的函数对象类，并说明如何将其有效地用作算法的回调。

第 23 章分析了 STL 编程的一些高级方面，并且特别强调了库的定制和扩展。其中介绍了如何使用和编写分配器、如何使用迭代器适配器、如何编写算法、如何编写容器以及如何编写迭代器。

尽管本章以及后两章会深入介绍 STL 的内容，但是标准模板库实在是太庞大了，要在这本书中详尽地介绍全部内容是不太可能的。本书网站上提供的资源中包含了标准库大多有用部分的参考。标准库头文件（Standard Library Header File）提供了标准库中所有头文件的一个总结，而标准库参考（Standard Library Reference）提供了 STL 中各个类和算法的一个参考。应该阅读第 21 章～第 23 章来了解 STL，但是要记住，这里并没有提到各个类所提供的每一个方法和成员，也没有显示每一个算法的原型。有关详细内容请参考附录。

### 21.1 容器概述

应该记得，第 4 章谈到，STL 中的容器是一些用于存储数据集合的通用数据结构。如果使用 STL，很少需要使用 C 风格的数组，也不需要自己来编写一个链表，或者设计一个栈。这些容器已经实现为模板，可以针对任何满足以下基本条件的类型对模板实例化。

STL 提供了 11 个容器，可划分为 4 类。顺序容器（sequential container）包括 vector（动态数组）、list 和 deque。关联容器（associative container）包括 map、multimap、set 和 multiset。容器适配器（container adapter）包括 queue、priority\_queue 和 stack。最后一个容器 bitset 本身作为一类。另外，C 风格的数组、C++ string 和流都可以在一定程度上用作为 STL 容器。

C++中的哪些容器可以作为 STL 中的一部分，对这个问题存在一些争议。这本书在这个定义上相对来讲包容性更大一些。有些人认为，只有顺序容器和关联容器才算得上 STL 的一部分。另外一些人尽管同意把字符串纳入，但是不认为 bitset 和容器适配器也能算进来。

在我们看来，容器是 STL 中最有价值的部分（不过，有些 C++ 支持者可能不同意这种看法）。如果没有足够的时间详细学习 STL，或者对此不感兴趣，那么至少应该考虑学学容器。一旦了解了一些语法细节，这些容器的学习就不再困难，而且还能在以后节省大量调试时间。

STL 中的所有内容都在 std 命名空间中。这本书中的例子总是会在源文件中使用这么一条语句：using namespace std;，不过，在自己的程序中，对于要使用 std 中的哪些符号，可以有自己的更多选择。

### 21.1.1 元素需求

STL 容器对元素使用的是值语义（value semantics）。也就是说，容器会存储所提供元素的一个副本，并在需求时返回这些元素的副本。容器还可以利用赋值操作符对元素赋值，以及用析构函数撤销元素。因此，在编写一个想要用于 STL（容器）的类时，要保证程序中完全可以同时有该对象的多个副本。

如果更喜欢引用语义，就必须通过保存元素的指针而不是元素本身，自行实现这种基于引用语义的容器。容器复制一个指针时，其结果仍指向同一个元素。

如果在容器中存储指针，建议你使用引用计数的智能指针，以便正确地处理内存管理。不过，不能在容器中使用 C++ auto\_ptr 类，因为它不能正确地实现复制（就 STL 而言）。在 STL 容器中使用 SuperSmartPointer 类，有关内容请见第 25 章。

表 21-1 列出了对容器中元素的具体需求。

表 21-1

方 法	描 述	说 明
复制构造函数	创建一个“等于”老元素的新元素，不过，这个新元素可以安全地撤销，而不会影响老元素	每次插入元素时都会用到
赋值操作符	用源元素的一个副本来代替一个元素的内容	每次修改元素时都会用到
析构函数	清除一个元素	每次删除元素时都会用到
默认构造函数	不带任何参数地构造一个元素	只对某些操作是必要的，如向量的 resize() 方法和映射的 operator [] 访问
operator ==	比较两个元素的相等性	只对某些操作是必要的，如对两个容器的 operator ==
operator <	确定一个元素是否比另一个元素小	只对某些操作是必要的，如对两个容器的 operator <。operator < 也是关联容器中对键的默认比较操作

STL 容器会经常对元素调用复制构造函数和赋值操作符，所以这些操作一定要高效。

对于如何编写这些方法，请参考第 9 章和第 16 章了解有关细节。

### 21.1.2 异常和错误检查

STL 容器提供了有限的错误检查。客户总想确保容器的使用是合法的。不过，有些容器方法和函数

会在某些条件下（如越界索引时）抛出异常。这一章将在适当的地方提到异常。本书网站上的标准库参考资源（Standard Library reference）会尽力列出每个方法可能抛出的异常。不过，要想全面地列出这些方法可能抛出的异常不太可能，因为这些方法是针对用户指定的类型来完成操作，而用户指定的类型有哪些异常特性预先并不可知。

### 21.1.3 迭代器

如第 4 章所述，STL 使用迭代器模式来提供一种通用的抽象，用以访问容器的元素。每个容器都提供了一个容器特定的迭代器，这是一个“美化”的智能指针，它知道如何迭代处理该特定容器的元素。所有不同容器的迭代器都遵循 C++ 中定义的一个特定接口。因此，即使容器提供不同的功能，对于想要使用容器元素的代码来说，迭代器可以为这些代码提供一个公共的接口。

可以把迭代器认为是容器中特定元素的一个指针。类似于数组中元素的指针，迭代器可以利用 `operator++` 移至下一个元素（译者注：即前向移动，forward）。类似地，可以对迭代器使用 `operator*` 和 `operator-->` 来访问具体的元素或元素的字段。有些迭代器允许用 `operator==` 和 `operator!=` 完成比较，而且支持 `operator--` 移至前面的元素（译者注：即反向移动，backward）。不同的容器提供的迭代器功能上稍有不同。标准定义了 5 类迭代器，表 21-2 对此做了总结。

表 21-2

迭代器种类	支持的操作	注 释
输入	<code>operator++</code> <code>operator*</code> <code>operator--&gt;</code> 复制构造函数 <code>operator==</code> <code>operator!=</code>	提供只读访问，仅为前向（没有 <code>operator--</code> 来反向移动）。（译者注：forward 和 backward 原义为前向和后向，这里的前向是指移到下一个元素，而后向是指移到前一个元素，为了避免混淆，在此把 backward 理解为与前向相对的反向）。可以用赋值操作符和复制构造函数来赋值和复制迭代器。可以比较迭代器的相等性
输出	<code>operator++</code> <code>operator*</code> 复制构造函数	提供只写访问，仅为前向。不能赋值或复制迭代器。不能比较迭代器的相等性。注意这里没有 <code>operator--&gt;</code>
前向	<code>operator++</code> <code>operator*</code> <code>operator--&gt;</code> 复制构造函数 默认构造函数 <code>operator==</code> <code>operator!=</code> <code>operator </code>	提供读/写访问，仅为前向。可以用赋值操作符和复制构造函数来赋值和复制迭代器。可以比较迭代器的相等性
双向	前向迭代器的功能，再加上： <code>operator--</code>	提供了前向迭代器所提供的所有功能。迭代器还可以反向移到前面的元素
随机访问	双向迭代器的功能，再加上： <code>operator+</code> ， <code>operator-</code> ， <code>operator+=</code> ， <code>operator-=</code> <code>operator&lt;</code> ， <code>operator&gt;</code> ， <code>operator&lt;=</code> ， <code>operator&gt;=</code> <code>operator[]</code>	等同于哑指针：迭代器支持指针运算、数组索引语法，以及各种形式的比较

提供迭代器的标准容器都配备有随机访问或双向访问迭代器。迭代器会重载所需的特定操作符，从这个意义上讲，迭代器的实现类似于智能指针类。有关操作符重载的详细内容请参见第 16 章，第 23 章

提供的一个示例迭代器实现。

基本迭代器的操作类似于哑指针支持的操作，所以哑指针可以作为某些容器的合法迭代器。实际上，vector 迭代器通常就只是实现为一个哑指针。不过，作为容器的客户，无需你操心这些实现细节，只需使用迭代器抽象就行了。

迭代器在内部可能并没有实现为指针，因此，在谈到通过迭代器可访问的元素时，我们会用“指示”（refer to）一词而不是“指向”（point to）。

第 22 章和第 23 章将更为深入地讨论迭代器和使用迭代器的 STL 算法。本章只是介绍使用各容器迭代器的基本知识。

只有顺序容器和关联容器才提供迭代器。容器适配器和位集不支持对元素的迭代处理。

### 公共迭代器类型定义和方法

STL 中每个支持迭代器的容器类为迭代器类型提供了公共的 typedef，名为 iterator 和 const\_iterator。这样一来，客户就可以使用容器迭代器而无需操心具体的类型。

const\_iterator 对容器的元素提供了只读访问。

容器还提供了一个 begin() 方法，这个方法会返回指示容器中第一个元素的迭代器。end() 方法会返回元素序列中最后元素之后的值（past-the-end）的一个引用。也就是说，end() 会返回一个迭代器，这个迭代器等同于对指示了序列中最后一个元素的一个迭代器应用 operator++ 的结果。begin() 和 end() 加在一起，提供了一个半开区间，即包括第一个元素，但不包括最后一个元素。这显然有些复杂，之所以引入这种复杂性是为了支持空区间（不包括任何元素的容器），对此 begin() 等于 end()。由迭代器 start 和 end 限定的半开区间通常以数学方式写作 [start, end)。（译者注：作者的意思是这个半开区间确实包括了容器中的所有元素，由于 end 指示的是最后一个元素之后的一个“元素”，因此半开区间不包括最后的这个虚元素。）

半开区间的概念还适用于传递给某些容器方法的迭代器区间，如 insert() 和 erase()。有关详细内容请见这一章对特定容器的描述。

## 21.2 顺序容器

vector、deque 和 list 都称为顺序容器，这是因为这些容器都以一种客户可见的顺序存储元素。要了解顺序容器，最好的办法就是深入到一个 vector 例子中，vector 也是最常使用的一种容器。这一节将以 vector 作为顺序容器的一个例子做详细描述，后面再简要介绍 deque 和 list。一旦熟悉了这些顺序容器，无论采用哪一种顺序容器都将非常简单。

### 21.2.1 vector

如第 4 章所述，STL vector 类似于一个数组，元素存储在一段连续的内存中，每个元素都位于自己的“槽”里。可以对 vector 索引访问，还可以在最后增加元素，或者是将元素插入到向量中的任何位置。向向量插入和从向量删除元素一般都需要线性时间，不过，有些操作实际上可以在 vector 最后位置上以摊分常量（amortized constant）时间完成，详细内容请见“向量内存分配机制”一节。随机访问单个元素的复杂性都为常量时间。

## 向量概述

向量在 `<vector>` 头文件中定义为一个带有两个类型参数的类模板：一个是在向量中存储的元素的类型，另一个参数是分配器（allocator）类型。

```
template <typename T, typename Allocator = allocator<T> > class vector;
```

Allocator 参数指定了客户可以设置的内存分配器的类型，以便使用定制内存分配。此模板参数有一个默认值，它使用了元素类型参数 T。有关模板参数的详细内容请参见第 11 章。

Allocator 类型参数的默认值对于大多数应用来说都足够了。程序员并不一定会觉得定制分配器有用，不过如果感兴趣，第 23 章还提供了有关的更多细节。这一章假设总是使用默认分配器。

## 定长向量

使用 vector 最简单的方法是将向量用作一个定长数组。向量提供了一个构造函数，可以指定元素的个数，并且提供了重载的 `operator[]` 来访问和修改这些元素。

就像是“真正的”数组索引一样，向量 `operator[]` 也没有提供越界检查。

例如，以下是一个小程序，可以对测验分数“规范化”，使最高分设置为 100，所有分数都相应地调整。这个程序创建了一个包括 10 个 double 的 vector，从用户处读入 10 个值，将各个值除最高分数（并乘以 100），然后打印出新值。为简单起见，这个程序中没有加错误检查。

```
#include <vector>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    vector<double> doubleVector(10); // Create a vector of 10 doubles.
    double max;
    int i;

    for (i = 0; i < 10; i++) {
        doubleVector[i] = 0;
    }

    // Read the first score before the loop in order to initialize max.
    cout << "Enter score 1: ";
    cin >> doubleVector[0];
    max = doubleVector[0];

    for (i = 1; i < 10; i++) {
        cout << "Enter score " << i + 1 << ": ";
        cin >> doubleVector[i];
        if (doubleVector[i] > max) {
            max = doubleVector[i];
        }
    }

    max /= 100;
    for (i = 0; i < 10; i++) {
```



```

        doubleVector[i] /= max;
        cout << doubleVector[i] << " ";
    }
    cout << endl;
    return (0);
}

```

从这个例子能够看出，可以像使用数组一样地使用向量。

向量的 `operator[]` 通常返回一个元素引用，可以把这个引用用在赋值语句的左边。如果在一个 `const vector` 对象上调用 `operator[]`，它会返回一个 `const` 元素的引用，此引用不能用作赋值的目标。有关如何实现这一点，详细内容请参见第 16 章。

### 指定一个初始元素值

在创建 `vector` 时，可以为元素指定一个初始值，如下所示：

```

#include <vector>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    vector<double> doubleVector(10, 0); // Creates vector of 10 doubles of value 0
    double max;
    int i;

    // No longer need to initialize each element: the ctor did it for you.

    // Read the first score before the loop in order to initialize max.
    cout << "Enter score 1: ";
    cin >> doubleVector[0];
    max = doubleVector[0];

    for (i = 1; i < 10; i++) {
        cout << "Enter score " << i + 1 << ": ";
        cin >> doubleVector[i];
        if (doubleVector[i] > max) {
            max = doubleVector[i];
        }
    }

    max /= 100;
    for (i = 0; i < 10; i++) {
        doubleVector[i] /= max;
        cout << doubleVector[i] << " ";
    }
    cout << endl;
    return (0);
}

```

### 其他向量元素访问方法

除了使用 `operator[]` 外，还可以通过 `at()`、`front()` 和 `back()` 等方法访问 `vector` 元素。`at()` 几乎与 `operator[]` 等同，只不过它会完成越界检查，如果索引越界，它会抛出一个 `out_of_range` 异常。`front()` 和 `back()` 分别返回 `vector` 中第一个和最后一个元素的引用。

访问任何 vector 元素的时间复杂性都是常量函数（即  $O(1)$ ）。

### 不定长向量

vector 真正的强大之处在于它能够动态增长。例如，还是考虑前面的测验分数“规范化”程序，不过现在有一个新的需求，它要处理任意多个测验分数。以下是这个新版本的程序：

```
int main(int argc, char** argv)
{
    vector<double> doubleVector; // Create a vector with zero elements.
    double max, temp;
    size_t i;

    // Read the first score before the loop in order to initialize max.
    cout << "Enter score 1: ";
    cin >> max;
    doubleVector.push_back(max);

    for (i = 1; true; i++) {
        cout << "Enter score " << i + 1 << " (-1 to stop): ";
        cin >> temp;
        if (temp == -1) {
            break;
        }
        doubleVector.push_back(temp);
        if (temp > max) {
            max = temp;
        }
    }

    max /= 100;
    for (i = 0; i < doubleVector.size(); i++) {
        doubleVector[i] /= max;
        cout << doubleVector[i] << " ";
    }
    cout << endl;
    return (0);
}
```

这个版本的程序使用了默认构造函数来创建一个包含 0 个元素的 vector。读入每个分数后，它会通过 push\_back() 方法增加到 vector 中。push\_back() 会负责为这个新元素分配空间。需要注意，最后一个 for 循环对 vector 使用了 size() 方法来确定容器中元素的个数。size() 返回一个无符号整数，所以为了做到兼容，i 的类型修改为 size\_t。

### 向量的有关细节

需要说明，你已经浅尝了 vector 的强大功能，下面要深入到它的细节当中了。

#### 构造函数和析构函数

默认构造函数创建一个包含 0 个元素的 vector。

```
#include <vector>
using namespace std;
```

```
int main(int argc, char** argv)
{
    vector<int> intVector; // Creates a vector of ints with zero elements
    return (0);
}
```

前面已经看到了，可以指定一些元素，还可以为这些元素指定一个值，如下所示：

```
#include <vector>
using namespace std;

int main(int argc, char** argv)
{
    vector<int> intVector(10, 100); // Creates a vector of 10 ints with value 100
    return (0);
}
```

如果不加默认值，新对象就会初始化为 0。如第 11 章所述，0 初始化（zero initialization）会用默认构造函数构造对象，并把诸如 int 和 double 等基本类型初始化为 0。

还可以如下创建内置类的向量：

```
#include <vector>
#include <string>
using namespace std;

int main(int argc, char** argv)
{
    vector<string> stringVector(10, "hello");
    return (0);
}
```

最后，还可以创建用户自定义类的向量：

```
#include <vector>
using namespace std;

class Element
{
public:
    Element() {}
    ~Element() {}
};

int main(int argc, char** argv)
{
    vector<Element> elementVector;
    return (0);
}
```

vector 存储的是对象的副本，其析构函数会调用各个对象的析构函数。

还可以在堆中分配向量：

```
#include <vector>
using namespace std;
```

```
class Element
{
public:
    Element() {}
    ~Element() {}
};

int main(int argc, char** argv)
{
```

```
    vector<Element>* elementVector = new vector<Element>(10);
    delete elementVector;
    return (0);
}
```

要记住，用 new 分配的向量用完后一定要调用 delete！

要使用 delete 而不是 delete[] 来释放向量。尽管 vector 实现为一个数组，但是所释放的只是向量对象。向量会自行处理底层数组。

### 向量复制和赋值

vector 类的复制构造函数和赋值操作符会对向量中所有元素完成深复制。因此，为了提高效率，应当按引用或 const 引用向函数或方法传递 vector。对于如何编写取模板实例化为参数的函数，有关细节请见第 11 章。

为了实现正常的复制和赋值，vector 提供了一个 assign() 方法，以便删除所有当前元素，并增加任意多个新元素。如果想重用一個 vector，这个方法就很有用。以下是一个简单的例子：

```
vector<int> intVector(10, 0);
// Other code . . .
intVector.assign(5, 100);
```

向量还提供了一个 swap() 方法，允许交换两个 vector 的内容。以下是一个简单例子：

```
vector<int> vectorOne(10, 0);
vector<int> vectorTwo(5, 100);

vectorOne.swap(vectorTwo);
// vectorOne now has 5 elements with the value 100.
// vectorTwo now has 10 elements with the value 0.
```

### 向量比较

STL 为 vector 提供了通常的 6 个重载比较操作符：==、!=、<、>、<= 和 >=。如果两个 vector 元素个数相同，而且两个 vector 中所有相应元素都相等，这两个 vector 就相等。两个 vector 比较时，如果第一个向量的 0 到 i-1 元素都等于第二个向量中的 0 到 i-1 元素，但是第一个向量中的 i 元素小于第二个向量中的 i 元素，则称第一个向量小于第二个向量。

用 operator== 或 operator!= 比较两个向量，这要求各元素可以用 operator== 来比较。用 operator<、operator>、operator<= 或 operator>= 比较两个向量时，要求各元素可以用 operator< 比较。如果想把一个定制类的对象存储在向量里，一定要编写这些操作符（译者注：即要为该类编写 operator== 和 operator< 操作符）。

以下是一个例子，这个简单程序完成了两个 int vector 的比较：

```
#include <vector>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    vector<int> vectorOne(10, 0);
    vector<int> vectorTwo(10, 0);

    if (vectorOne == vectorTwo) {
        cout << "equal!\n";
    } else {
        cout << "not equal!\n";
    }

    vectorOne[3] = 50;

    if (vectorOne < vectorTwo) {
        cout << "vectorOne is less than vectorTwo\n";
    } else {
        cout << "vectorOne is not less than vectorTwo\n";
    }
    return (0);
}
```

这个程序的输出如下：

```
equal!
vectorOne is not less than vectorTwo
```

#### 向量迭代器

本章开始在“迭代器”一节解释过容器迭代器的基本知识。那时的讨论有一点抽象，具体来看一个代码例子会有帮助。以下还是测验分数规范化的程序，不过先前使用 size() 的 for 循环现在替换为使用一个迭代器的 for 循环。

```
#include <vector>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    vector<double> doubleVector;
    double max, temp;
    int i;

    // Read the first score before the loop in order to initialize max.
    cout << "Enter score 1: ";
    cin >> max;
    doubleVector.push_back(max);

    for (i = 1; true; i++) {
        cout << "Enter score " << i + 1 << " (-1 to stop): ";
```

```

    cin >> temp;
    if (temp == -1) {
        break;
    }
    doubleVector.push_back(temp);
    if (temp > max) {
        max = temp;
    }
}
max /= 100;
for (vector<double>::iterator it = doubleVector.begin();
     it != doubleVector.end(); ++it) {
    *it /= max;
    cout << *it << " ";
}
cout << endl;
return (0);
}

```

可以看到这个新例子中的 for 循环有许多 STL 代码。首先，请看 for 循环初始化语句：

```
vector<double> :: iterator it = doubleVector.begin();
```

应该记得每个容器都定义了一个名为 `iterator` 的类型来表示该类容器的迭代器。`begin()` 返回指示容器中第一个元素的该类迭代器。因此，初始化语句在变量 `it` 中包含一个指示 `doubleVector` 中第一个元素的迭代器。接下来，来看 for 循环比较：

```
it != doubleVector.end();
```

这个语句只是检查迭代器是否走过了 `vector` 中的全部元素序列。如果到达这一点（即过了最后一个元素），循环终止。自增语句 `++it` 会让迭代器自增，使之指示 `vector` 中的下一个元素。

应该尽可能使用先自增而不是后自增，因为先自增至少与后自增同样高效，而且往往更为高效。`it++` 必须返回一个新的迭代器对象，而 `++it` 可以只返回对象的一个引用。有关 `operator++` 实现的详细内容请见第 16 章，编写迭代器的有关细节请参见第 23 章。

for 循环体包括两行代码：

```

    *it /= max;
    cout << *it << " ";

```

可以看到，这个代码可以在迭代处理时同时访问和修改元素。第一行使用 `*it` 解除引用来得到它所指示的元素，并为该元素赋值。第二行再次对 `it` 解除引用，不过，这一次只是将元素以流的方式输出至 `cout`。

### 访问对象元素的字段

如果容器的元素是对象，可以对迭代器使用 `->` 操作符来调用或访问这些对象的成员。例如，以下小程序创建了一个包括 10 个 `string` 的 `vector`，然后迭代处理所有元素，为每个元素追加一个新 `string`：

```

#include <vector>
#include <string>
using namespace std;

```





学习在线

视频资料下载  
电子书交流

[www.eimhe.com](http://www.eimhe.com)



学习在线

视频资料下载  
电子书交流

[www.eimhe.com](http://www.eimhe.com)

```
int main(int argc, char** argv)
{
    vector<string> stringVector(10, "hello");

    for (vector<string>::iterator it = stringVector.begin();
        it != stringVector.end(); ++it) {
        it->append("there");
    }
}
```

#### const\_iterator

常规的 iterator 是可读写的。不过，如果对一个 const 对象调用 begin() 和 end()，都会得到一个 const\_iterator。const\_iterator 是只读的，不能修改元素。iterator 总是可以转换为一个 const\_iterator，因此编写如下的代码肯定是安全的：

```
vector<type>::const_iterator it = myVector.begin();
```

不过，const\_iterator 不能转换为一个 iterator。如果 myVector 是 const，以下代码无法通过编译：

```
vector<type>::iterator it = myVector.begin();
```

因此，如果不需要修改 vector 的元素，就应该使用一个 const\_iterator。这条规则能让代码更为通用。

#### 迭代器安全

一般来讲，迭代器与指针的安全程度几乎一样，都极其不安全。例如，可以编写如下的代码：

```
vector<int> intVector;

vector<int>::iterator it = intVector.end();
*it = 10; // BUG! it doesn't refer to a valid element.
```

还记得 end() 返回的迭代器已经越过了 vector 的最后元素（即它指示的是最后一个元素之后的位置），对其解除引用没有明确的定义，这通常意味着程序会崩溃。不过，迭代器本身不需要完成任何验证。

记住，end() 返回的迭代器会越过容器的最后一个元素，而不是返回指示了容器最后一个元素的迭代器。

如果使用了不匹配的迭代器，还会出现另一个问题。例如，以下代码从 vectorTwo 初始化一个迭代器，并尝试将其与 vectorOne 的末尾（end）迭代器比较。不用多说，这个循环并不会像你原想的那样工作，它永远也不会终止。

```
vector<int> vectorOne(10);
vector<int> vectorTwo(10);

// Fill in the vectors.

// BUG! Infinite loop
for (vector<int>::iterator it = vectorTwo.begin(); it != vectorOne.end();
    ++it) {
    // Loop body
}
```

### 其他迭代器操作

向量迭代器是随机访问的，这说明可以前向或后向移动，或者跳至某个元素。例如，以下代码最后会把 vector 中第 5 个元素（索引为 4）的值修改为 4：

```
vector<int> intVector(10, 0);

vector<int>::iterator it = intVector.begin();
it += 5;
--it;
*it = 4;
```

第 23 章会对不同类的迭代器提供更多信息。

### 迭代器 vs 索引

可以写一个使用简单索引变量的 for 循环，并用 size() 方法迭代处理 vector 中的元素，既然如此，为什么还要那么麻烦地使用迭代器呢？这个问题问得好，对此有 3 个主要原因。

- 利用迭代器，可以在容器中的任意位置上插入和删除元素及元素序列。请见以下“增加和删除元素”一节。
- 利用迭代器，可以使用 STL 算法，有关 STL 算法请见第 22 章。
- 与对容器索引从而单个地获得各个元素相比，使用一个迭代器顺序地访问各个元素往往更为高效。这种一般性对于 vector 并不成立，但是适用于 list、map 和 set。

### 增加和删除元素

你已经了解了，可以用 push\_back() 方法向 vector 追加一个元素。vector 提供了一个与之相应的删除方法 pop\_back()。

pop\_back() 并不返回所删除的元素。如果想得到这个元素，必须先用 back() 获得该元素。

还可以用 insert() 方法在向量中的任意位置插入元素。insert() 会向迭代器指定的一个位置增加一个或多个元素，它会让所有后续的元素后移，以便为新插入的元素腾出空间。有 3 种不同形式的重载 insert()：其中一个 insert() 可以插入一个元素，另一个可以插入同一个元素的 n 个副本，还有一个可以从一个迭代器区间插入元素。应该记得，迭代器区间是半开区间，因此，它包括开始迭代器指示的元素，但是不包括末尾迭代器指示的元素。

push\_back() 和 insert() 取元素的 const 引用，根据需要分配内存来存储新元素，并保存元素参数的副本。

类似地，可以利用 erase() 从 vector 中任意位置删除元素。有两种形式的 erase()，可以删除单个的元素，也可以删除由迭代器指定的一个区间。利用 clear() 可以删除所有元素。

以下是一个小程序，这个程序展示了增加和删除元素的各个方法。它使用了一个辅助函数 printVector()，可以把向量的内容打印到 cout，不过这个函数的实现未在这里提供，因为它使用了第 22 章和第 23 章才会介绍的算法。

```
int main(int argc, char** argv)
{
    vector<int> vectorOne, vectorTwo;
    int i;
```

```
vectorOne.push_back(1);
vectorOne.push_back(2);
vectorOne.push_back(3);
vectorOne.push_back(5);

// Oops, we forgot to add 4. Insert it in the correct place.
vectorOne.insert(vectorOne.begin() + 3, 4);

// Add elements 6 through 10 to vectorTwo.
for (i = 6; i <= 10; i++) {
    vectorTwo.push_back(i);
}

printVector(vectorOne);
printVector(vectorTwo);

// Add all the elements from vectorTwo to the end of vectorOne.
vectorOne.insert(vectorOne.end(), vectorTwo.begin(), vectorTwo.end());

printVector(vectorOne);

// Clear vectorTwo entirely.
vectorTwo.clear();

// And add 10 copies of the value 100.
vectorTwo.insert(vectorTwo.begin(), 10, 100);

// Decide we only want 9 elements.
vectorTwo.pop_back();

// Now erase the numbers 2 through 5 in vectorOne.
vectorOne.erase(vectorOne.begin() + 1, vectorOne.begin() + 5);

printVector(vectorOne);
printVector(vectorTwo);

return (0);
}
```

这个程序的输出如下：

```
1 2 3 4 5
6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 6 7 8 9 10
100 100 100 100 100 100 100 100 100
```

应该记得，迭代器提供了一个半开区间，`insert()` 会在迭代器位置所指元素的前面增加新元素。因此，可以把整个 `vectorTwo` 的内容如下插入到 `vectorOne` 的后面：

```
vectorOne.insert (vectorOne.end(), vectorTwo.begin(), vectorTwo.end());
```

诸如 `insert()` 和 `erase()` 等方法（取一个 `vector` 区间为参数）认为开始和末尾迭代器指示的是同一个容器中的元素，而且末尾迭代器指示的元素要么在开始迭代器所指示元素的后面，要么指示开始迭代器所指示的同一个元素。如果不满足这些前置条件，这些方法就不能正常工作。

### 算法复杂性和迭代器无效

在 `vector` 中插入或删除元素会导致所有后续元素上移或下移，以便为新插入的元素腾出空间，或者填补所删除元素留下的洞。因此，这些操作的时间复杂性是线性的。另外，在此操作之后，所有指示插入或删除点或后续位置的迭代器都是无效的。迭代器并没有“魔法般地”跟着元素在 `vector` 中上移或下移。

还要记住，内部的 `vector` 重分配会导致指示向量中元素的所有迭代器都会无效，而不只是那些指示插入点或删除点之后元素的迭代器无效。有关详细内容请见下面的“向量内存分配机制”小节。

### 向量内存分配机制

`vector` 会自动地分配内存来存储所插入的元素。应该记得，`vector` 的需求指出，所有元素都必须放在连续的内存中，就像是 C 风格的数组。因为它无法请求在当前内存块的后面增加内存，每次 `vector` 分配更多内存时，它都必须在另外一个内存位置分配一个新的更大的内存块，并且把所有元素复制到这个新的内存块。这个过程很耗费时间，因此向量实现力图尽量避免这个过程，即在必须完成重新分配时，它会分配超过需要的更多空间。采用这种做法，就可以避免每次插入一个元素时都必须重新分配内存。

这里有一个很明显的问题，作为向量的一个客户，为什么还要操心向量在内部如何管理内存呢？你可能认为，基于抽象原则，应该不用考虑 `vector` 内存分配机制的内部细节了。遗憾的是，你确实需要理解其工作原理，对此有两点原因。

1. 效率。`vector` 分配机制可以保证元素插入能够在摊分常量时间 (amortized constant time) 内完成，此操作大多都会在常量时间内完成，不过偶尔（如果需要重新分配）会是线性时间。如果看重效率，就可以在 `vector` 完成重新分配时有所控制。

2. 迭代器无效。重新分配会使所有指示 `vector` 中元素的迭代器都无效。因此，利用 `vector` 接口，可以查询和控制 `vector` 重新分配。如果没有显式地控制重新分配，就应该有心理准备，要假设所有插入都可能导致重新分配，而且由此会导致所有迭代器失效。

### 大小和容量

向量提供了两个方法来得到其大小的信息：`size()` 和 `capacity()`。`size()` 会返回 `vector` 中的元素个数，而 `capacity()` 返回的是不重新分配的情况下最多能容纳的元素个数。因此，在不导致重新分配的情况下，所能插入的元素个数就是 `capacity() - size()`。

可以用 `empty()` 方法来查询一个 `vector` 是否为空。`vector` 可以为空，但是这个 `vector` 容量可以非 0。

### 预留容量

如果不担心效率或迭代器失效，就没有必要显式地控制 `vector` 内存分配。不过，如果想让程序尽可能地高效，或者想保证迭代器不会失效，就可以要求 `vector` 预先分配足够多的空间来保存所有元素。当然，需要知道它要保存多个元素，而有时这是无法预知的。

预先分配空间的一种方法是调用 `reserve()`。这个方法会分配足够多的内存来保存指定数目的元素。“向量示例：循环调度类”小节将提供 `reserve()` 方法的一个实战例子。

为元素预留空间会改变容量，但是不会改变大小。也就是说，它并没有真正创建元素。不要访问越过 `vector` 大小的元素。

预先分配空间的另一种方法是在构造函数中指定希望该 `vector` 存放多少个元素。这个方法会实际创建一个有此大小的 `vector`（或者有此容量的 `vector`）。



### 向量示例：循环调度类

在一组有限资源中分配请求，这是计算机科学领域的一个常见问题。例如，网络负载均衡器要在可以为请求提供服务的不同主机之间分配到来的网络连接。理想情况下，每个主机的忙闲程度都应该一样。对于这个问题，最简单的算法解决方案之一就是循环调度（round-robin scheduling），采用这种循环调度，会轮流使用或处理资源。用完最后一个资源后，调度程序又会从第一个资源重新开始。例如，对于一个带 3 个主机的网络负载均衡器，第一个请求会交给第一个主机，第二个请求会交给第二个主机，第三个请求则交给第三个主机，第四个请求又交给第一个主机。如此周而复始，无限循环下去。

假设你决定编写一个通用的循环调度类，可用于任何类型的资源。这个类支持增加和删除资源，而且增加和删除资源可以无限进行，另外应该支持资源的循环处理以便得到下一个资源。可以直接使用 STL vector，不过，编写一个包装类从而更直接地提供特定应用所需的功能，这往往很有好处。以下例子显示了一个 RoundRobin 类模板，并且提供了相关注释来解释有关代码。首先，下面给出的是类定义：

```
#include <stdexcept>
#include <vector>
using std::vector;

//
// Class template RoundRobin
//
// Provides simple round-robin semantics for a list of elements.
// Clients add elements to the end of the list with add().
//
// getNext() returns the next element in the list, starting with the first,
// and cycling back to the first when the end of the list is reached.
//
// remove() removes the element matching the argument.
//
template <typename T>
class RoundRobin
{
public:
    //
    // Client can give a hint as to the number of expected elements for
    // increased efficiency.
    //
    RoundRobin(int numExpected = 0);
    ~RoundRobin();

    //
    // Appends elem to the end of the list. May be called
    // between calls to getNext().
    //
    void add(const T& elem);

    //
    // Removes the first (and only the first) element
    // in the list that is equal (with operator==) to elem.
    // May be called between calls to getNext().
    //
    void remove(const T& elem);

    //

```

```

    // Returns the next element in the list, starting from 0 and continuously
    // cycling, taking into account elements that are added or removed.
    //
    T& getNext() throw(std::out_of_range);
protected:
    vector<T> mElems;
    typename std::vector<T>::iterator mCurElem;
private:
    // Prevent assignment and pass-by-reference.
    RoundRobin(const RoundRobin& src);
    RoundRobin& operator=(const RoundRobin& rhs);
};

```

可以看到，这个公共接口很简单，只有 3 个方法，再外加一个构造函数和析构函数。资源存储在向量 `mElems` 中。`mCurElem` 是一个迭代器，总是指示 `mElems` 中下一个要用于循环调度机制的元素。要注意这里在声明 `mCurElem` 的代码行前面使用了关键字 `typename`。到此为止，只是见过这个关键字可以用于指定模板参数，不过这个关键字还有一个用处。如果要基于一个或多个模板参数访问一个类型，就必须显式地指定 `typename`。在这个例子中，要用模板参数 `T` 访问 `iterator` 类型。因此，必须指定 `typename`。C++ 往往有一些奇怪的语法，这就是这些奇怪语法中的一个例子。

以下是 `RoundRobin` 类的实现。要注意构造函数中使用了 `reserve()`，另外在 `add()`、`remove()` 和 `getNext()` 中大量使用了迭代器。这里最难的是保持 `mCurElem` 一直有效，即使在 `add()` 或 `remove()` 之后 `mCurElem` 也要指示当前元素。

```

template <typename T>
RoundRobin<T>::RoundRobin(int numExpected)
{
    // If the client gave a guideline, reserve that much space.
    mElems.reserve(numExpected);

    // Initialize mCurElem even though it isn't used until
    // there's at least one element.
    mCurElem = mElems.begin();
}

template <typename T>
RoundRobin<T>::~RoundRobin()
{
    // Nothing to do here--the vector will delete all the elements
}

//
// Always add the new element at the end.
//
template <typename T>
void RoundRobin<T>::add(const T& elem)
{
    //
    // Even though we add the element at the end,
    // the vector could reallocate and invalidate the iterator.
    // Take advantage of the random access iterator features to save our
    // spot.
    //
    int pos = mCurElem - mElems.begin();
}

```

```

// Add the element.
mElems.push_back(elem);

// If it's the first element, initialize the iterator to the beginning.
if (mElems.size() == 1) {
    mCurElem = mElems.begin();
} else {
    // Set it back to our spot.
    mCurElem = mElems.begin() + pos;
}
}

template <typename T>
void RoundRobin<T>::remove(const T& elem)
{
    for (typename std::vector<T>::iterator it = mElems.begin(); it != mElems.end();
        ++it) {
        if (*it == elem) {
            //
            // Removing an element will invalidate our mCurElem iterator if
            // it refers to an element past the point of the removal.
            // Take advantage of the random access features of the iterator
            // to track the position of the current element after the removal.
            //
            int newPos;

            // If the current iterator is before or at the one we're removing,
            // the new position is the same as before.
            if (mCurElem <= it) {
                newPos = mCurElem - mElems.begin();
            } else {
                // Otherwise, it's one less than before.
                newPos = mCurElem - mElems.begin() - 1;
            }
            // Erase the element (and ignore the return value).
            mElems.erase(it);

            // Now reset our iterator.
            mCurElem = mElems.begin() + newPos;

            // If we were pointing to the last element and it was removed,
            // we need to loop back to the first.
            if (mCurElem == mElems.end()) {
                mCurElem = mElems.begin();
            }
            return;
        }
    }
}

template <typename T>
T& RoundRobin<T>::getNext() throw (std::out_of_range)
{
    // First, make sure there are any elements.
    if (mElems.empty()) {
        throw std::out_of_range("No elements in the list");
    }

    // Retrieve a reference to return.
    T& retVal = *mCurElem;

```

```

// Increment the iterator modulo the number of elements.
++mCurElem;
if (mCurElem == mElems.end()) {
    mCurElem = mElems.begin();
}

// Return the reference.
return (retVal);
}

```

以下是负载均衡器的一个简单实现，它使用了 RoundRobin 类模板。这里忽略了具体的网络代码，因为网络代码与具体的操作系统有关。

```

#include "RoundRobin.h"

//
// Forward declaration for NetworkRequest
// Implementation details omitted
//
class NetworkRequest;

//
// Simple Host class that serves as a proxy for a physical machine.
// Implementation details omitted.
//
class Host
{
public:
    //
    // Implementation of processRequest would forward
    // the request to the network host represented by the
    // object. Omitted here.
    //
    void processRequest(NetworkRequest& request) {}
};

//
// Simple load balancer that distributes incoming requests
// to its hosts using a round-robin scheme
//
class LoadBalancer
{
public:
    //
    // Constructor takes a vector of hosts.
    //
    LoadBalancer(const vector<Host>& hosts);
    ~LoadBalancer() {}

    //
    // Ship the incoming request to the next host using
    // a round-robin scheduling algorithm.
    //
    void distributeRequest(NetworkRequest& request);
};

```



```
protected:
    RoundRobin<Host> rr;
};

LoadBalancer::LoadBalancer(const vector<Host>& hosts)
{
    // Add the hosts.
    for (size_t i = 0; i < hosts.size(); ++i) {
        rr.add(hosts[i]);
    }
}

void LoadBalancer::distributeRequest(NetworkRequest& request)
{
    try {
        rr.getNext().processRequest(request);
    } catch (out_of_range& e) {
        cerr << "No more hosts.\n";
    }
}
```

### 21.2.2 vector<bool> 特殊化

标准要求 vector 为 bool 实现一个部分特殊化 (partial specialization)，其目的是通过“压缩”布尔值来优化空间分配。应该记得，bool 要么为 true，要么为 false，因此只需 1 位就可以表示，1 位正好可以代表两个值。不过，大多数 C++ 编译器都让 bool 与 int 的大小相同。vector<bool> 就是要以一个位的方式来存储“bool 数组”，由此来节省空间。

可以把 vector<bool> 认为是一个位域，而不是一个 vector。相对于 vector<bool>，后面介绍的 bitset 容器提供了一个功能更完备的位域实现。不过，vector<bool> 有自己的好处，这就是它可以动态地改变大小。

由于考虑到要为 vector<bool> 提供一些位域例程，所以这里有一个额外的方法：flip()。这个方法可以在容器上调用，在这种情况下它会对容器中的所有元素取反，也可以在 operator [] 或类似方法返回的一个引用上调用此方法，此时它会把该元素取反。

在此，你可能想知道如何对一个 bool 引用调用方法。这个问题的答案是根本不能调用。vector<bool> 特殊化实际上定义了一个名为 reference 的类来作为底层 bool（或位）的一个代理。在调用 operator []、at() 或一个类似方法时，vector<bool> 会返回一个 reference 对象，这是实际 bool 的一个代理。

vector<bool> 返回的引用实际上是代理，这说明不能取其地址来得到容器中实际元素的指针。代理设计模式将在第 26 章详细介绍。

在实际中，通过压缩 bool 只能节省很少的空间，而这与为此付出的额外努力来说，往往得不偿失。不过，因为在此有额外的 flip() 方法，而且引用实际上是代理对象，所以应该熟悉这种部分实例化的做法。许多 C++ 专家建议避免使用 vector<bool>，而应采用 bitset，除非确实需要一个变长的位域。

### 21.2.3 deque

deque 与 vector 几乎完全相同，不过用得很少。这二者之间的主要区别在于：

- 实现不必把元素连续地存储在内存中。
- deque 在队头和队尾插入和删除元素时都为常量时间（vector 只是在最后位置插入和删除元素时才为摊分常量时间）。
- deque 提供了 push\_front() 和 pop\_front()，这是 vector 所没有的。
- deque 没有通过 reserve() 或 capacity() 提供内存管理机制。

你的应用很少需要 deque，这与 vector 或 list 不同。因此，我们只是在本书网站上标准库参考资源中列出了 deque 方法的有关细节。

### 21.2.4 list

STL list 是一个标准双向链表。在列表中任意位置插入和删除元素时都为常量时间，不过访问各个元素较慢（为线性时间）。实际上，列表甚至没有提供诸如 operator [] 的随机访问操作。只能通过迭代器才能访问单个元素。

大多数 list 操作都与 vector 的相应操作相同，包括构造函数、析构函数、复制操作、赋值操作和比较操作。这一节将强调与 vector 不同的方法。这里没有谈到的 list 方法有关细节请参见本书网站上标准库参考资源。

#### 访问元素

list 对于访问元素只提供了两个方法：front() 和 back()。这两个方法的运行时间都为常量。所有其他元素的访问都必须通过迭代器完成。

列表没有提供对元素的随机访问。

#### 迭代器

list 迭代器是双向的，但不像 vector 迭代器那样能随机访问。这说明不能对 list 迭代器加减，也不能对其完成其他的指针运算。

#### 增加和删除元素

list 与 vector 一样，也支持同样的元素增加和删除方法，包括 push\_back()、pop\_back()、三种形式的 insert()、两种形式的 erase() 和 clear()。

类似于 deque，它也提供了 push\_front() 和 pop\_front()。有关列表的特异之处在于，一旦找到了正确的位置，所有这些方法（除了 clear()）都在常量时间运行。因此，如果应用要从数据结构完成多次插入和删除操作，而且不需要快速的基于索引的元素访问，列表就很适合。

#### 列表大小

类似于 deque，但不同于 vector，list 没有提供底层内存模型。因此，列表支持 size() 和 empty()，但不支持 resize() 或 capacity()。

#### 特殊的列表操作

list 提供了一些特殊的操作，这些操作充分利用了快速的元素插入和删除。这一节将提供一个概述和一些例子。本书网站上的标准库参考资源提供了所有方法的全面参考。

#### 接合

list 类具有链表的特性，这就使它可以在一个 list 中的任何位置以常量时间接合 (splice) 或插入另一



个完整的 list。可以如下使用这个方法的最简单的版本：

```
#include <list>
#include <string>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    list<string> dictionary, bWords;

    // Add the a words.
    dictionary.push_back("aardvark");
    dictionary.push_back("ambulance");
    dictionary.push_back("archive");

    // Add the c words.
    dictionary.push_back("canticle");
    dictionary.push_back("consumerism");
    dictionary.push_back("czar");

    // Create another list, of the b words.
    bWords.push_back("bathos");
    bWords.push_back("balderdash");
    bWords.push_back("brazen");

    // Splice the b words into the main dictionary.
    list<string>::iterator it;
    int i;

    // Iterate up to the spot where we want to insert bs
    // for loop body intentionally empty--we're just moving up three elements.
    for (it = dictionary.begin(), i = 0; i < 3; ++it, ++i);

    // Add in the bwords. This action removes the elements from bWords.
    dictionary.splice(it, bWords);
    // Print out the dictionary.
    for (it = dictionary.begin(); it != dictionary.end(); ++it) {
        cout << *it << endl;
    }

    return (0);
}
```

运行这个程序得到的结果如下：

```
aardvark
ambulance
archive
bathos
balderdash
brazen
canticle
consumerism
czar
```

另外还有两种形式的 `splice()`：一个 `splice()` 是从另一个 `list` 插入一个元素，还有一个 `splice()` 是从另一个 `list` 插入一个区间。有关详细内容请见本书网站上标准库参考资源。

接合对于作为参数传入的 `list` 是破坏性的：它会从一个 `list` 中删除要接合的元素，使之插入到另一个 `list` 中。

### 更高效的算法版本

除了 `splice()`，列表类还为许多通用 STL 算法提供了特殊实现。这些算法的通用形式将在第 22 章中介绍。在此只讨论 `list` 提供的特殊版本。

如果可以选择，应使用 `list` 的方法而不是通用算法，因为前者更为高效。

`list` 为一些算法提供了特殊实现作为 `list` 的方法，表 21-3 对这些算法做了总结。可以参考本书网站上标准库参考资源和第 22 章来了解有关原型、算法的详细内容，以及在 `list` 上调用这些算法的特定运行时间。

表 21-3

方 法	说 明
<code>remove()</code> <code>remove_if()</code>	从 <code>list</code> 删除某些元素
<code>unique()</code>	从 <code>list</code> 删除重复的连续元素
<code>merge()</code>	合并两个 <code>list</code> 。开始时必须对两个 <code>list</code> 排序 类似于 <code>splice()</code> ， <code>merge()</code> 对于作为参数传入的 <code>list</code> 也具有破坏性
<code>sort()</code>	对 <code>list</code> 中的元素完成一个稳定排序
<code>reverse()</code>	对 <code>list</code> 中的元素逆置

以下程序展示了上述大多数方法。

### 列表示例：确定注册

假设你在为一所大学编写一个计算机注册系统。你要提供的一个特性是必须能够根据每个班的学生列表生成大学所有注册学生的完整列表。对于这个例子，假设必须只写一个函数，此函数取两个参数，一个参数是学生名 `list` 的 `vector`，学生名表示为 `string`，另一个参数是因为没有交学费中断课程学习的学生列表。这个方法应当生成所有课程中所有学生的一个完整 `list`，其中没有重复，而且不包括已经中断学习的学生。需要注意，学生可能会参与多门课程的学习。

以下是这个方法的代码。借助于 STL `list` 的强大功能，这个方法很简短，反不如它的说明多。要注意，STL 允许容器“嵌套”：在这个例子中，可以使用 `list` 的 `vector`。

```
#include <list>
#include <vector>
#include <string>
using namespace std;

//
// classLists is a vector of lists, one for each course. The lists
// contain the students enrolled in those courses. They are not sorted.
//
// droppedStudents is a list of students who failed to pay their
```

```

// tuition and so were dropped from their courses.
//
// The function returns a list of every enrolled (nondropped) student in
// all the courses.
//
list<string>
getTotalEnrollment(const vector<list<string> >& classLists,
    const list<string>& droppedStudents)
{
    list<string> allStudents;

    // Concatenate all the course lists onto the master list.
    for (size_t i = 0; i < classLists.size(); ++i) {
        allStudents.insert(allStudents.end(), classLists[i].begin(),
            classLists[i].end());
    }

    // Sort the master list.
    allStudents.sort();

    // Remove duplicate student names (those who are in multiple courses).
    allStudents.unique();

    //
    // Remove students who are on the dropped list.
    // Iterate through the dropped list, calling remove on the
    // master list for each student in the dropped list.
    //
    for (list<string>::const_iterator it = droppedStudents.begin();
        it != droppedStudents.end(); ++it) {
        allStudents.remove(*it);
    }

    // Done!
    return (allStudents);
}

```

## 21.3 容器适配器

除了三个共享的顺序容器，STL 还提供了三个容器适配器：queue、priority\_queue 和 stack。这些适配器都是包装了某个顺序容器的包装器。其目的是简化接口，并且只提供适合于 stack 或 queue 抽象的特性。例如，适配器没有提供迭代器，也不能同时插入或删除多个元素。

容器适配器的接口也许不能满足你的需求。倘若如此，可以直接使用顺序容器，或者编写自己的、功能更为完备的适配器。有关适配器设计模式的详细内容请参见第 26 章。

### 21.3.1 queue

queue 容器适配器在头文件 <queue> 中定义，提供了标准的“先进先出”（FIFO）语义。如以往一样，这个容器适配器写作为一个类模板，形式如下。

```
template <typename T, typename Container = deque<T> > class queue;
```

T 模板参数指定你要在 queue 中存储的类型。通过第二个模板参数可以规定 queue 适配的底层容器。不过，队列要求这个底层顺序容器既支持 push\_back()，又支持 pop\_front()，因此只有两个内置的顺序容器可供选择：deque 和 list。对大多数用途来说，可以只使用默认的 deque。

### 队列操作

queue 接口相当简单，只有 6 个方法，外加一个构造函数和常规的比较操作符。push() 方法会在队尾增加一个新元素。pop() 从队头删除元素。可以分别利用 front() 和 back() 获取第一个和最后一个元素的引用，但不删除相应元素。与平常一样，在 const 对象上调用时，front() 和 back() 会返回 const 引用，在非 const 对象上调用时，这两个方法会返回非 const（可读写）引用。

pop() 并不返回弹出的元素。如果想保留一个副本，就必须先用 front() 获得队头元素。

队列还支持 size() 和 empty()。详细内容请参见本书网站上的标准库参数资源。

### 队列示例：一个网络数据包缓冲区

两个计算机通过网络通信时，他们相互之间发送的信息会划分为具体的块，这称为数据包（packet，也称分组）。计算机操作系统的网络层必须提取这些数据包，并在其到达时保存起来。不过，计算机可能没有足够的带宽来一次全部处理。因此，网络层通常会缓冲（buffer）或保存这些数据包，直到更高层有机会注意到它们。数据包应当按到达的顺序进行处理，因此这个问题特别适合采用 queue 结构。以下是一个小的 PacketBuffer 类，它会在一个 queue 中保存到来的数据包，直到这些数据包得到处理。这是一个模板，因此网络的不同层可以使用这个模板来处理不同类型的数据包，如 IP 数据包或 TCP 数据包。它允许客户指定一个最大的大小，因为操作系统通常会限制可以保存的数据包个数，以保证不至于使用过多内存。当缓冲区满时，将忽略后面到来的数据包。

```
#include <queue>
#include <stdexcept>
using std::queue;

template <typename T>
class PacketBuffer
{
public:
    //
    // If maxSize is nonpositive, the size is unlimited.
    // Otherwise only maxSize packets are allowed in
    // the buffer at any one time.
    //
    PacketBuffer(int maxSize = -1);

    //
    // Stores the packet in the buffer.
    // Throws overflow_error is the buffer is full.
    //
    void bufferPacket(const T& packet);

    //
    // Returns the next packet. Throws out_of_range
    // if the buffer is empty.
    //
    T getNextPacket() throw (std::out_of_range);

protected:
```

```

    queue<T> mPackets;
    int mMaxSize;

private:
    // Prevent assignment and pass-by-value.
    PacketBuffer(const PacketBuffer& src);
    PacketBuffer& operator=(const PacketBuffer& rhs);
};

template <typename T>
PacketBuffer<T>::PacketBuffer(int maxSize)
{
    mMaxSize = maxSize;
}

template <typename T>
void PacketBuffer<T>::bufferPacket(const T& packet)
{
    if (mMaxSize > 0 && mPackets.size() ==
        static_cast<size_t>(mMaxSize)) {
        // No more space. Just drop the packet.
        return;
    }

    mPackets.push(packet);
}

template <typename T>
T PacketBuffer<T>::getNextPacket() throw (std::out_of_range)
{
    if (mPackets.empty()) {
        throw (std::out_of_range("Buffer is empty"));
    }
    // Retrieve the head element.
    T temp = mPackets.front();
    // Pop the head element.
    mPackets.pop();
    // Return the head element.
    return (temp);
}

```

这个类的实际应用可能需要多线程。不过，以下例子类似于一个单元测试，用以展示这个类的应用：

```

#include "PacketBuffer.h"
#include <iostream>
using namespace std;

class IPPacket {};

int main(int argc, char** argv)
{
    PacketBuffer<IPPacket> ipPackets(3);

    ipPackets.bufferPacket(IPPacket());
    ipPackets.bufferPacket(IPPacket());
}

```



```
ipPackets.bufferPacket(IPPacket());
ipPackets.bufferPacket(IPPacket());
while (true) {
    try {
        IPPacket packet = ipPackets.getNextPacket();
    } catch (out_of_range&) {
        cout << "Processed all packets!" << endl;
        break;
    }
}
return (0);
}
```

### 21.3.2 priority\_queue

priority\_queue 也是一个队列，其元素按有序顺序排列。在此并不采用严格的 FIFO 顺序，给定时刻位于队头的元素正是有最高优先级的元素，这个元素可能是队列中最老的元素，也可能是最新的元素。如果两个元素有相同的优先级，那么它们在队列中的顺序就遵循 FIFO 语义。

STL priority\_queue 容器适配器也定义在头文件 <queue> 中。其模板定义如下所示（在此稍有简化）。

```
template <typename T, typename Container = vector<T>, typename Compare =
    less<T> >;
```

这看上去很复杂，实际上没那么可怕！在此之前，我们已经见过前面的两个参数：T 是 priority\_queue 中存储的元素类型，Container 是 priority\_queue 适配的底层容器。priority\_queue 默认地使用 vector，不过也可以使用 deque。list 不能用，因为 priority\_queue 要求能对元素随机访问以便进行排序。第三个参数 Compare 要难一些。在第 22 章将更多地了解到，less 是一个类模板，它利用 operator< 支持对两个类型为 T 的对象的比较。对你而言，这意味着 queue 中元素优先级是根据 operator< 确定的。可以定制所用的比较，不过这是第 22 章的内容。就目前来说，只要确保为 priority\_queue 中存储的类型适当地定义了 operator< 就可以。

优先队列的队头元素是有“最高”优先级的元素，默认地优先级根据 operator< 确定，这样如果某些元素“小于”另外一些元素，则前者优先级较低。

#### 优先队列操作

与 queue 相比，priority\_queue 提供的操作更少。push() 和 pop() 允许分别插入和删除元素，top() 会返回队头元素的一个 const 引用。

即使在一个非 const 对象上调用，top() 也会返回一个 const 引用。priority\_queue 没有提供机制来获得队尾元素。pop() 并不返回所弹出的元素。如果想保留一个副本，必须先用 top() 获取队头元素。

类似于 queue，priority\_queue 支持 size() 和 empty()。不过，它没有提供任何比较操作符。有关详细内容请参见本书网站上的标准库参数资源。

这个接口很受限。具体地，priority\_queue 没有提供迭代器支持，而且不可能合并两个 priority\_queue。



### 优先队列示例：错误关联器

系统上的一个故障通常会导致不同组件生成多个错误。好的错误处理系统会使用错误关联（error correlation）来避免处理重复的错误，而且可以保证最先处理最重要的错误。可以使用 `priority_queue` 编写一个非常简单的错误关联器。这个类只是根据事件的优先级对事件排序，这样最高优先级的错误总是最先处理。以下是其类定义：

```
#include <ostream>
#include <string>
#include <queue>
#include <stdexcept>

// Sample Error class with just a priority and a string error description
class Error
{
public:
    Error(int priority, std::string errMsg) :
        mPriority(priority), mError(errMsg) {}
    int getPriority() const {return mPriority;}
    std::string getErrorString() const {return mError;}

    friend bool operator<(const Error& lhs, const Error& rhs);
    friend std::ostream& operator<<(std::ostream& str, const Error& err);

protected:
    int mPriority;
    std::string mError;
};

// Simple ErrorCorrelator class that returns highest priority errors first
class ErrorCorrelator
{
public:
    ErrorCorrelator() {}

    //
    // Add an error to be correlated.
    //
    void addError(const Error& error);

    //
    // Retrieve the next error to be processed.
    //
    Error getError() throw (std::out_of_range);

protected:
    std::priority_queue<Error> mErrors;
private:
    // Prevent assignment and pass-by-reference.
    ErrorCorrelator(const ErrorCorrelator& src);
    ErrorCorrelator& operator=(const ErrorCorrelator& rhs);
};
```

以下是函数和方法的定义：

```

#include "ErrorCorrelator.h"
using namespace std;

bool operator<(const Error& lhs, const Error& rhs)
{
    return (lhs.mPriority < rhs.mPriority);
}

ostream& operator<<(ostream& str, const Error& err)
{
    str << err.mError << " (priority " << err.mPriority << ")";
    return (str);
}

void ErrorCorrelator::addError(const Error& error)
{
    mErrors.push(error);
}

Error ErrorCorrelator::getError() throw (out_of_range)
{
    //
    // If there are no more errors, throw an exception.
    //
    if (mErrors.empty()) {
        throw (out_of_range("No elements!"));
    }

    // Save the top element.
    Error top = mErrors.top();
    // Remove the top element.
    mErrors.pop();
    // Return the saved element.
    return (top);
}

```

下面是一个简单的单元测试，用以展示如何使用 ErrorCorrelator。实际的使用可能需要多线程，这样就能让一个线程增加错误，另一些线程处理这些错误。

```

#include "ErrorCorrelator.h"
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    ErrorCorrelator ec;

    ec.addError(Error(3, "Unable to read file"));
    ec.addError(Error(1, "Incorrect entry from user"));
    ec.addError(Error(10, "Unable to allocate memory!"));

    while (true) {
        try {
            Error e = ec.getError();
            cout << e << endl;
        }
    }
}

```

```

    } catch (out_of_range& ) {
        cout << "Finished processing errors\n";
        break;
    }
}

return (0);
}

```

### 21.3.3 stack

stack 与 queue 基本相同，只不过栈提供了“后进先出”（LIFO）语义，而不是 FIFO。其模板定义如下所示。

```
template <typename T, typename Container = deque<T> > class stack;
```

可以使用三个标准顺序容器中的任何一个作为 stack 的底层模型。

#### 栈操作

类似于 queue，stack 提供了 push() 和 pop()。其区别在于，push() 会在栈顶增加一个新元素，把所有先前插入的元素“向下压”，pop() 则从栈顶删除元素，这也是最近插入的元素。top() 方法在一个 const 对象上调用时，会返回一个 const 引用，如果在一个非 const 对象上调用，将返回一个非 const 引用。

pop() 不会返回所弹出的元素。如果想保留一个副本，必须先用 top() 获得栈顶元素。

stack 支持 empty()、size() 和标准比较操作符。有关详细内容请参见本书网站上的标准库参数资源。

#### 栈示例：修改后的错误关联器

假设你决定重写先前的 ErrorCorrelator 类，使之给出最新的错误，而不是有最高优先级的错误。只需用一个 stack 来替换 ErrorCorrelator 类定义中的 priority\_queue。现在，Error 会从类按 LIFO 顺序分发，而不是按优先级顺序。在方法定义中，并不需要做任何修改，因为 push()、pop()、top() 和 empty() 方法在 priority\_queue 和 stack 中都有。

```

#include <ostream>
#include <string>

#include <stack>
#include <stdexcept>

// Details of Error class omitted for brevity

//
// Simple ErrorCorrelator class that returns most recent errors first
//
class ErrorCorrelator
{
public:
    ErrorCorrelator() {}

    //
    // Add an error to be correlated.

```

```

//
void addError(const Error& error);

//
// Retrieve the next error to be processed.
//
Error getError() throw (std::out_of_range);

protected:
    std::stack<Error> mErrors;

private:
    // Prevent assignment and pass-by-reference.
    ErrorCorrelator(const ErrorCorrelator& src);
    ErrorCorrelator& operator=(const ErrorCorrelator& rhs);
};

```

## 21.4 关联容器

不同于顺序容器，关联容器并不在线性配置中存储元素。相反，它们提供了一个键到值的映射。一般地，关联容器的插入、删除和查找时间都相同。

STL 提供的 4 个关联容器包括：map、multimap、set 和 multiset。这些容器都将元素存储在一个有序的、类似于树的数据结构中。

### 21.4.1 pair 工具类

在学习关联容器之前，必须先熟悉 pair 类，这个类定义在 <utility> 头文件中。pair 是一个类模板，它将两个值组织在一起，这两个值的类型可能不同。可以通过 first 和 second 公共数据成员来访问这两个值。为 pair 定义了 operator== 和 operator< 来比较 first 和 second 元素。

以下是一些例子。

```

#include <utility>
#include <string>
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    // Two-argument ctor and default ctor
    pair<string, int> myPair("hello", 5), myOtherPair;

    // Can assign directly to first and second
    myOtherPair.first = "hello";
    myOtherPair.second = 6;

    // Copy ctor.
    pair<string, int> myThirdPair(myOtherPair);

    // operator<
    if (myPair < myOtherPair) {
        cout << "myPair is less than myOtherPair\n";
    } else {

```

```

        cout << "myPair is greater than or equal to myOtherPair\n";
    }

    // operator==
    if (myOtherPair == myThirdPair) {
        cout << "myOtherPair is equal to myThirdPair\n";
    } else {
        cout << "myOtherPair is not equal to myThirdPair\n";
    }

    return (0);
}

```

标准库还提供了一个工具函数模板 `make_pair()`，它能从两个变量构造一个 `pair`。例如，可以如下使用这个工具函数模板：

```
pair<int, int> aPair = make_pair(5, 10);
```

当然，在这种情况下，可以只使用两参数的构造函数。不过，如果想把一个 `pair` 传递给一个函数，`make_pair()` 将更有用。不同于类模板，函数模板可以从参数推导出类型，因此可以使用 `make_pair()` 来构造一个 `pair`，而无需显式地指定类型。

在 `pair` 中使用指针类型很危险，因为 `pair` 复制构造函数和赋值操作符只完成指针类型的浅复制和赋值。

### 21.4.2 map

`map` 是最有用的容器之一。它存储的是键/值对而不是一个值。插入、查找和删除都基于键完成，值只是“顺带的”。“映射”(`map`)一词就源于其概念理解，即容器将键“映射”至值。你可能对散列表的概念更熟悉一些。映射也提供了一个类似的接口，只是在底层数据结构和操作的算法复杂性上存在区别。

`map` 会基于元素的键来保证元素有序，因此插入、删除和查找都取对数时间。通常 `map` 实现为某种形式的平衡树，如红黑树。不过，客户并不会看到这个树结构。

如果要基于一个“键”值来存储和获取元素，就应该使用 `map`。

#### 构造映射

`map` 模板取 4 个类型：键类型、值类型、比较类型和分配器类型。与以往一样，这一章先不谈分配器，有关的详细内容请见第 23 章。比较类型类似于上述 `priority_queue` 的比较类型。利用比较类型，可以指定一个与默认比较类不同的比较类。通常无需改变排序规则。在这一章中，我们只使用默认的 `less` 比较。使用默认比较类时，要确保键都适当地对应于 `operator<`。

如果对更多细节感兴趣，第 22 章解释了如何编写自己的比较类。

如果忽略了比较和分配器参数（在此就要求你这样做），构造 `map` 就与构造一个 `vector` 或 `list` 颇为相似，只不过要在模板中分别指定键和值类型。例如，以下代码会构造一个 `map`，这个映射使用 `int` 作为键，并保存 `Data` 类的对象作为值（在此没有提供其完整定义）：

```

#include <map>
using namespace std;

class Data

```



```

{
    public:
        Data(int val = 0) { mVal = val; }
        int getVal() const { return mVal; }
        void setVal(int val) {mVal = val; }
        // Remainder of definition omitted
    protected:
        int mVal;
};

int main(int argc, char** argv)
{
    map<int, Data> dataMap;
    return (0);
}

```

### 插入元素

向诸如 vector 和 list 等顺序容器插入元素时，总是要指定要在哪里增加元素。map 以及其他关联容器则与此不同。map 内部实现会确定保存新元素的位置，你要做只是提供键和值。

map 和其他关联容器确实提供了一个取迭代器位置的 insert()。不过，这个位置只是对容器的一个正确位置“提示”。容器并不一定非要在该位置上插入元素。

在插入元素时，要记住重要的一点，map 支持所谓的“惟一键”。map 中的每个元素都必须有一个不同的键。如果想支持多个元素有相同的键，就必须使用 multimap，这在后面介绍。

向 map 中插入元素有两种方法，一种比较笨，另一种没有那么笨。

#### insert() 方法

向 map 中增加一个元素的笨方法是 insert() 方法。对此，一个问题是必须指定键/值对作为一个 pair 对象。第二个问题是，基本形式 insert() 的返回值是 iterator 和 bool 的一个对 (pair)。为什么返回这么复杂的返回值，原因是，如果已经有指定键的元素，insert() 不会重写（覆盖）这个元素的值。所返回 pair 的 bool 元素会指示 insert() 是否确实插入了新的键/值对。iterator 会指示 map 中有指定键的元素（可能为新值，也可能是原来的值，这取决于插入是否成功）。继续前面的 map 例子，在此说明如何使用 insert()：

```

#include <map>
#include <iostream>
using namespace std;

class Data
{
    public:
        Data(int val = 0) { mVal = val; }
        int getVal() const { return mVal; }
        void setVal(int val) {mVal = val; }
        // Remainder of definition omitted
    protected:
        int mVal;
};

int main(int argc, char** argv)

```



```

{
    map<int, Data> dataMap;
    pair<map<int, Data>::iterator, bool> ret;

    ret = dataMap.insert(make_pair(1, Data(4)));
    if (ret.second) {
        cout << "Insert succeeded!\n";
    } else {
        cout << "Insert failed!\n";
    }

    ret = dataMap.insert(make_pair(1, Data(6)));
    if (ret.second) {
        cout << "Insert succeeded!\n";
    } else {
        cout << "Insert failed!\n";
    }
    return (0);
}

```

需要注意，这里使用了 `make_pair()` 来构造 `pair`，以便传递给 `insert()` 方法。这个程序的输出如下：

```

Insert succeeded!
Insert failed!

```

`operator []`

向 `map` 插入元素还有一个不那么笨的方法，这就是通过重载的 `operator []`。主要是在语法上有区别：要分别指定键和值。另外 `operator []` 总能成功。如果不存在有给定键的元素值，它会用该键和值创建一个新的元素。如果已经存在有给定键的元素，`operator []` 会把现有的元素值替换为新指定的值。下面还是前面的例子，只不过这里使用了 `operator []` 而不是 `insert()`：

```

#include <map>
#include <iostream>
using namespace std;

class Data
{
public:
    Data(int val = 0) { mVal = val; }
    int getVal() const { return mVal; }
    void setVal(int val) { mVal = val; }
    // Remainder of definition omitted
protected:
    int mVal;
};

int main(int argc, char** argv)
{
    map<int, Data> dataMap;
    dataMap[1] = Data(4);
    dataMap[1] = Data(6); // Replaces the element with key 1
    return (0);
}

```

不过，对于 operator [] 有一点警告：它总是会构造一个新的值对象，即使并不需要使用这个对象。因此，要求元素值（值对象）有一个默认构造函数，而且这种做法可能没有 insert() 的效率高。

### map 迭代器

map 迭代器的工作与顺序容器迭代器的工作很类似。主要区别在于，这里的迭代器指示的是键/值对而不只是一个值。要想访问值，必须获取 pair 对象的 second 字段。下面展示了如何从前一个例子迭代处理 map：

```
#include <map>
#include <iostream>
using namespace std;

class Data
{
public:
    Data(int val = 0) { mVal = val; }
    int getVal() const { return mVal; }
    void setVal(int val) { mVal = val; }
    // Remainder of definition omitted
protected:
    int mVal;
};

int main(int argc, char** argv)
{
    map<int, Data> dataMap;

    dataMap[1] = Data(4);
    dataMap[1] = Data(6); // Replaces the element with key 1

    for (map<int, Data>::iterator it = dataMap.begin();
         it != dataMap.end(); ++it) {
        cout << it->second.getVal() << endl;
    }
    return (0);
}
```

再来看访问值的表达式：

it->second.getVal()

it 指示了一个键/值 pair，所以可以使用 -> 操作符来访问该 pair 的 second，这是一个 Data 对象。然后可以在该数据对象上调用 getVal() 方法。

需要注意以下代码与上面的表达式功能相当：

```
(*it).second.getVal()
```

你还会看到许多类似的代码，因为迭代器一般不使用 ->。

可以通过非 const 迭代器修改元素值，但是不能修改元素的键，即使是使用一个非 const 迭代器也不允许。这是因为修改键会破坏 map 中元素的有序顺序。

map 迭代器是双向的。

### 查找元素

映射基于所提供的键可以在对数时间内完成元素查找。如果你已经知道映射中有给定键的元素，查找这个元素最简单的方法就是通过 operator []。operator [] 的好处是它会返回一个元素，可以直接使用（如果是非 const 映射，还可以修改）该元素引用，而不必操心要把值从 pair 对象中取出来。以下是对前一个例子的扩展，在此对 Data 对象调用了 setVal() 方法，并指定键为 1。

```
#include <map>
#include <iostream>
using namespace std;

class Data
{
public:
    Data(int val = 0) { mVal = val; }
    int getVal() const { return mVal; }
    void setVal(int val) { mVal = val; }
    // Remainder of definition omitted
protected:
    int mVal;
};

int main(int argc, char** argv)
{
    map<int, Data> dataMap;
    dataMap[1] = Data(4);
    dataMap[1] = Data(6);
    dataMap[1].setVal(100);

    return (0);
}
```

不过，如果不知道元素是否存在，可能不想使用 operator []，因为如果没有找到有给定键的元素，它会基于这个键插入一个新元素。还有一种做法，map 提供了一个 find() 方法，如果存在有指定键的元素，它会返回指示这个元素的一个 iterator，如果 map 中不存在这样的元素，就会返回 end iterator（译者注：即末尾迭代器）。以下是使用 find() 的一个例子，在此基于键 1 对 Data 对象完成同样的修改。

```
#include <map>
#include <iostream>
using namespace std;

class Data
{
public:
    Data(int val = 0) { mVal = val; }
    int getVal() const { return mVal; }
    void setVal(int val) { mVal = val; }

    // Remainder of definition omitted
protected:
    int mVal;
};

int main(int argc, char** argv)
{

```

```

map<int, Data> dataMap;
dataMap[1] = Data(4);
dataMap[1] = Data(6);

map<int, Data>::iterator it = dataMap.find(1);
if (it != dataMap.end()) {
    it->second.setVal(100);
}

return (0);
}

```

可以看到，使用 `find()` 稍微麻烦一些，不过有时这很有必要。如果只想知道 `map` 中到底有没有一个有指定键的元素，可以使用 `count()` 成员函数。它会返回 `map` 中有给定键的元素的个数。对于 `map`，这个结果只能是 0 或 1，因为其中不存在键重复的元素。“删除元素”小节将给出一个使用 `count()` 的例子。

### 删除元素

`map` 允许在特定迭代器位置删除一个元素，或者删除一个给定迭代器区间中的所有元素，这两个操作的运行时间分别是摊分常量时间和对数时间。从客户的角度看，这两个 `erase()` 方法与顺序容器中的删除方法相当。不过，映射有一个突出的特性，这就是它还提供了另一个版本的 `erase()`，可以删除与一个键匹配的元素。

以下是一个例子。

```

// #includes, Data class definition, and beginning of main function omitted.
// See previous examples for details.
map<int, Data> dataMap;
dataMap[1] = Data(4);
cout << "There are " << dataMap.count(1) << " elements with key 1\n";
dataMap.erase(1);
cout << "There are " << dataMap.count(1) << " elements with key 1\n";

```

### 映射示例：银行账户

可以使用 `map` 来实现一个简单的银行账户。一种常见的模式是键作为一个 `class` 或 `struct` 的一个字段，而该 `class` 或 `struct` 存储在 `map` 中。在这个例子中，键就是账号。下面是简单的 `BankAccount` 和 `BankDB` 类：

```

#include <map>
#include <string>
#include <stdexcept>
using std::map;
using std::string;
using std::out_of_range;

class BankAccount
{
public:
    BankAccount(int acctNum, const string& name) :
        mAcctNum(acctNum), mClientName(name) {}
    void setAcctNum(int acctNum) { mAcctNum = acctNum; }
    int getAcctNum() const { return (mAcctNum); }
    void setClientName(const string& name) { mClientName = name; }
    string getClientName() const { return mClientName; }

```

```

        // Other public methods omitted

protected:
    int mAcctNum;
    string mClientName;
    // Other data members omitted
};

class BankDB
{
public:
    BankDB() {}

    // Adds acct to the bank database. If an account
    // exists already with that number, the new account is
    // not added. Returns true if the account is added, false
    // if it's not.
    bool addAccount(const BankAccount& acct);

    // Removes the account acctNum from the database
    void deleteAccount(int acctNum);

    // Returns a reference to the account represented
    // by its number or the client name.
    // Throws out_of_range if the account is not found
    BankAccount& findAccount(int acctNum) throw(out_of_range);
    BankAccount& findAccount(const string& name) throw(out_of_range);

    // Adds all the accounts from db to this database.
    // Deletes all the accounts in db.
    void mergeDatabase(BankDB& db);

protected:
    map<int, BankAccount> mAccounts;
};

```

以下是 BankDB 方法的实现。

```

#include "BankDB.h"
#include <utility>
using namespace std;

bool BankDB::addAccount(const BankAccount& acct)
{
    // Declare a variable to store the return from insert().
    pair<map<int, BankAccount>::iterator, bool> res;
    // Do the actual insert, using the account number as the key.
    res = mAccounts.insert(make_pair(acct.getAcctNum(), acct));

    // Return the bool field of the pair specifying success or failure.
    return (res.second);
}

void BankDB::deleteAccount(int acctNum)
{

```

```

    mAccounts.erase(acctNum);
}

BankAccount& BankDB::findAccount(int acctNum) throw(out_of_range)
{
    // Finding an element via its key can be done with find().
    map<int, BankAccount>::iterator it = mAccounts.find(acctNum);
    if (it == mAccounts.end()) {
        throw (out_of_range("No account with that number."));
    }
    // Remember that iterators into maps refer to pairs of key/value.
    return (it->second);
}

BankAccount& BankDB::findAccount(const string& name) throw(out_of_range)
{
    //
    // Finding an element by a non-key attribute requires a linear
    // search through the elements.
    //
    for (map<int, BankAccount>::iterator it = mAccounts.begin();
         it != mAccounts.end(); ++it) {
        if (it->second.getClientName() == name) {
            // Found it!
            return (it->second);
        }
    }
    throw (out_of_range("No account with that name."));
}

void BankDB::mergeDatabase(BankDB& db)
{
    // Just insert copies of all the accounts in the old db
    // into the new one.
    mAccounts.insert(db.mAccounts.begin(), db.mAccounts.end());

    // Now delete all the accounts in the old one.
    db.mAccounts.clear();
}

```

### 21.4.3 multimap

multimap 是一个允许有多个同键元素的 map。其接口与 map 接口基本相同，只有以下几点改变：

- multimap 没有提供 operator []。由于一个键可能对应多个元素，所以这个操作符没有意义。
- multimap 上的插入总会成功。因此，多映射 (multimap) insert() 增加一个元素时，并不需要返回 iterator 和 bool 的 pair。它只返回 iterator。

multimaps 允许插入相同的键/值对。如果想避免这种冗余性，必须在插入新元素之前先显式做检查。

multimap 最难的部分是查找元素。不能使用 operator []，因为没有提供这个操作符。find() 不是特别有用，因为它返回的 iterator 会指示有给定键的任何元素（不一定是该键的第一个元素）。

幸运的是，multimap 会把所有带相同键的元素存储在一起，而且提供了一些方法来得到容器中同键



元素子区间的相应 iterator。lower\_bound() 和 upper\_bound() 都返回一个 iterator，分别指示第一个元素和越过最后元素的“元素”。如果不存在与该键匹配的元素，则 lower\_bound() 和 upper\_bound() 返回的 iterator 相等。

如果不想分别调用两个方法来得到界定给定键元素的 iterator，multimap 还提供了一个 equal\_range() 方法，它会返回 lower\_bound() 和 upper\_bound() 所返回两个 iterator 的一个 pair。

下面的例子展示了这些方法的使用。

lower\_bound()、upper\_bound() 和 equal\_range() 方法在 map 中也有，但是用途很有限。

### 多映射示例：好友列表

大多数在线聊天程序允许用户有一个“好友列表”或朋友列表。聊天程序会对好友列表中列出的用户授予特权，如允许他们向用户发送主动消息。

为在线聊天程序实现好友列表的一种方法是把信息存储在一个 multimap 中。一个 multimap 可以存储所有用户的好友列表。容器中的每一项存储对应一个用户的好友。键是用户，值是好友。例如，如果这本书的两个作者都在对方的好友列表中，那么就会有形式如下的两项：“Nicholas Solter”映射至“Scott Kleper”和“Scott Kleper”映射至“Nicholas Solter”。multimap 允许同一个键有多个值，因此一个用户可以有多个好友。以下是 BuddyList 类定义。

```
#include <map>
#include <string>
#include <list>

using std::multimap;
using std::string;
using std::list;

class BuddyList
{
public:
    BuddyList();

    //
    // Adds buddy as a friend of name
    //
    void addBuddy(const string& name, const string& buddy);

    //
    // Removes buddy as a friend of name
    //
    void removeBuddy(const string& name, const string& buddy);

    //
    // Returns true if buddy is a friend of name.
    // Otherwise returns false.
    //
    bool isBuddy(const string& name, const string& buddy) const;

    //
    // Retrieves a list of all the friends of name
    //
    list<string> getBuddies(const string& name) const;
```

```
protected:
    multimap<string, string> mBuddies;
private:
    // Prevent assignment and pass-by-value.
    BuddyList(const BuddyList& src);
    BuddyList& operator=(const BuddyList& rhs);
};
```

以下是实现。它展示了 `lower_bound()`、`upper_bound()` 和 `equal_range()` 的使用。

```
#include "BuddyList.h"
using namespace std;

BuddyList::BuddyList()
{
}

void BuddyList::addBuddy(const string& name, const string& buddy)
{
    // Make sure this buddy isn't already there.
    // We don't want to insert an identical copy of the
    // key/value pair.
    if (!isBuddy(name, buddy)) {
        mBuddies.insert(make_pair(name, buddy));
    }
}

void BuddyList::removeBuddy(const string& name, const string& buddy)
{
    // Declare two iterators into the map.
    multimap<string, string>::iterator start, end;

    // Obtain the beginning and end of the range of elements with
    // key name. Use both lower_bound() and upper_bound() to demonstrate
    // their use. Otherwise, could just call equal_range().
    start = mBuddies.lower_bound(name);
    end = mBuddies.upper_bound(name);

    // Iterate through the elements with key name looking
    // for a value buddy.
    for (start; start != end; ++start) {
        if (start->second == buddy) {
            // We found a match! Remove it from the map.
            mBuddies.erase(start);
            break;
        }
    }
}

bool BuddyList::isBuddy(const string& name, const string& buddy) const
{
    // Declare two iterators into the map.
    multimap<string, string>::const_iterator start, end;
    // Obtain the beginning and end of the range of elements with
    // key name. Use both lower_bound() and upper_bound() to demonstrate
    // their use. Otherwise, could just call equal_range().
```

```

start = mBuddies.lower_bound(name);
end = mBuddies.upper_bound(name);

// Iterate through the elements with key name looking
// for a value buddy. If there are no elements with key name,
// start equals end, so the loop body doesn't execute.
for (start; start != end; ++start) {
    if (start->second == buddy) {
        // We found a match!
        return (true);
    }
}
// No matches
return (false);
}

list<string> BuddyList::getBuddies(const string& name) const
{
    // Create a variable to store the pair of iterators.
    pair<multimap<string, string>::const_iterator,
        multimap<string, string>::const_iterator> its;

    // Obtain the pair of iterators marking the range containing
    // elements with key name.
    its = mBuddies.equal_range(name);

    // Create a list with all the names in the range
    // (all the buddies of name).
    list<string> buddies;
    for (its.first; its.first != its.second; ++its.first) {
        buddies.push_back((its.first->second));
    }

    return (buddies);
}

```

要注意，removeBuddy() 不能只是使用“完全删除”版本的 erase()，即把有给定键的所有元素都删除，因为它应当只删除有给定键的一个元素，而非全部。还要注意，getBuddies() 不能使用 list 的 insert() 在 equal\_range() 返回的区间内插入元素，因为 multimap iterator 指示的元素是键/值对，而不是 string。getBuddies() 必须显式地迭代处理 list，从各个键/值对抽取出 string，并将其压入到待返回的新 list 中。

以下是 BuddyList 的一个简单测试。

```

#include "BuddyList.h"
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    BuddyList buddies;

    buddies.addBuddy("Harry Potter", "Ron Weasley");
    buddies.addBuddy("Harry Potter", "Hermione Granger");
    buddies.addBuddy("Harry Potter", "Hagrid");
    buddies.addBuddy("Harry Potter", "Draco Malfoy");
}

```

```
// That's not right! Remove Draco.
buddies.removeBuddy("Harry Potter", "Draco Malfoy");

buddies.addBuddy("Hagrid", "Harry Potter");
buddies.addBuddy("Hagrid", "Ron Weasley");
buddies.addBuddy("Hagrid", "Hermione Granger");

list<string> harryBuds = buddies.getBuddies("Harry Potter");

cout << "Harry's friends: \n";
for (list<string>::const_iterator it = harryBuds.begin();
     it != harryBuds.end(); ++it) {
    cout << "\t" << *it << endl;
}

return (0);
}
```

#### 21.4.4 set

set 容器与 map 非常类似。区别在于，集合不存储键/值对，set 中，值本身就是键。如果要存储没有显式键的信息，但是又希望对元素排序以便快速插入、查找和删除，此时 set 就很有用。

set 提供的接口与 map 的接口几乎相同。主要区别是 set 没有提供 operator []。另外，尽管标准中没有明确指出来，但是大多数实现都令 set iterator 等同于 const\_iterator，因此不能通过 iterator 来修改 set 的元素。即使你的 STL 版本允许通过一个 iterator 修改 set 元素，也要避免这样做，因为修改 set 中的元素（仍在容器中）会破坏有序顺序。

##### 集合示例：访问控制列表

在计算机系统上实现基本安全的一种做法是通过访问控制列表。系统上的每个实体（如一个文件或一个设备）都有相应的允许访问该实体的用户列表。用户一般只能由有特殊权限的用户从一个实体的许可列表中增加和删除用户。在内部，set 容器提供了一种很不错的方法来表示访问控制列表。可以对应每个实体有一个 set，其中包括允许访问该实体的所有用户名。下面是一个简单的访问控制列表的类定义。

```
#include <set>
#include <string>
#include <list>
using std::set;
using std::string;
using std::list;

class AccessList
{
public:
    AccessList() {}

    //
    // Adds the user to the permissions list.
    //
    void addUser(const string& user);

    //
}
```

```

// Removes the user from the permissions list
//
void removeUser(const string& user);

//
// Returns true if user is in the permissions list
//
bool isAllowed(const string& user) const;

//
// Returns a list of all the users who have permissions
//
list<string> getAllUsers() const;

protected:
    set<string> mAllowed;
};

```

以下是方法定义。

```

#include "AccessList.h"
using namespace std;

void AccessList::addUser(const string& user)
{
    mAllowed.insert(user);
}

void AccessList::removeUser(const string& user)
{
    mAllowed.erase(user);
}

bool AccessList::isAllowed(const string& user) const
{
    return (mAllowed.count(user) == 1);
}

list<string> AccessList::getAllUsers() const
{
    list<string> users;
    users.insert(users.end(), mAllowed.begin(), mAllowed.end());
    return (users);
}

```

最后，在此提供一个简单的测试程序。

```

#include "AccessList.h"
#include <iostream>
#include <iterator>
using namespace std;

int main(int argc, char** argv)
{
    AccessList fileX;
    fileX.addUser("nsolter");
}

```



```
fileX.addUser("klep");
fileX.addUser("baduser");
fileX.removeUser("baduser");

if (fileX.isAllowed("nsolter")) {
    cout << "nsolter has permissions\n";
}

if (fileX.isAllowed("baduser")) {
    cout << "baduser has permissions\n";
}

list<string> users = fileX.getAllUsers();
for (list<string>::const_iterator it = users.begin();
     it != users.end(); ++it) {
    cout << *it << " ";
}
cout << endl;

return (0);
}
```

#### 21.4.5 multiset

multiset 与 set 的关系就如同 multimap 与 map。multiset 支持 set 的所有操作，不过它允许容器中同时存储彼此相等的多个元素。需要注意，元素可能是对象，尽管这些对象并非同一对象，但用 operator== 比较是相等的。我们没有给出一个 multiset 的例子，因为它与 set 和 multimap 非常相似。

### 21.5 其他容器

前面已经提到，C++ 语言中还有其他一些方面在某种程度上与 STL 有关，这包括数组、string、流和 bitset。

#### 21.5.1 数组作为 STL 容器

应该记得，“哑”指针可以很好地作为迭代器，因为它们支持所需的操作符。这一点绝非小事。这说明你可以把常规的 C++ 数组当作 STL 容器，只需使用元素的指针作为迭代器。当然，数组并没有提供诸如 size()、empty()、insert() 和 erase() 等方法，因此它们不是真正的 STL 容器。不过，由于数组通过指针确实支持迭代器，因此可以在第 22 章所述的算法和这一章介绍的某些方法中使用数组。

例如，可以使用 vector insert() 方法将一个数组中的所有元素复制到一个 vector 中，此方法取任意容器的一个迭代器区间。insert() 方法原型如下所示。

```
template <typename InputIterator> void insert(iterator position,
                                             InputIterator first, InputIterator last);
```

如果想用一个 int 数组作为复制源，InputIterator 的模板化类型则为 int\*。以下是这个完整的例子。

```
#include <vector>
#include <iostream>
```



```
using namespace std;

int main(int argc, char** argv)
{
    int arr[10]; // normal C++ array
    vector<int> vec; // STL vector

    //
    // Initialize each element of the array to the value of
    // its index.
    //
    for (int i = 0; i < 10; i++) {
        arr[i] = i;
    }

    //
    // Insert the contents of the array into the
    // end of the vector.
    //
    vec.insert(vec.end(), arr, arr + 10);

    // Print the contents of the vector.
    for (i = 0; i < 10; i++) {
        cout << vec[i] << " ";
    }

    return (0);
}
```

需要注意，指示数组第一个元素的迭代器只是第一个元素的地址。还记得第 13 章中曾提到，数组名本身就解释为第一个元素的地址。指示最后位置的迭代器必须是越过最后一个元素的位置，因此这是第一个元素的地址加 10。

### 21.5.2 string 作为 STL 容器

可以把 string 看作是字符的一个顺序容器。因此，了解到 C++ string 是一个完备的顺序容器应该并不奇怪。它包括 begin() 和 end() 方法（会返回指向 string 内部的迭代器）、insert() 和 erase() 方法、size()、empty()，以及所有其他顺序容器的基本功能。这与 vector 相当相近，甚至还提供了 reserve() 和 capacity() 方法。不过，不同于 vector，string 不需要将元素连续地存储在内存中。另外 vector 提供的某些方法 string 并没有提供，如 push\_back()。

C++ string 实际上是 basic\_string 模板类 char 实例化的一个 typedef。不过，为简单起见，我们只谈到 string。在此有关 wstring 的讨论同样适用于 basic\_string 模板的其他实例化。

可以像使用 vector 一样将 string 用作一个 STL 容器。以下是一个例子。

```
#include <string>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    string str1;
```

```
    str1.insert(str1.end(), 'h');
    str1.insert(str1.end(), 'e');
    str1.insert(str1.end(), 'l');
    str1.insert(str1.end(), 'l');
    str1.insert(str1.end(), 'o');

    for (string::const_iterator it = str1.begin(); it != str1.end(); ++it) {
        cout << *it;
    }
    cout << endl;

    return 0;
}
```

除了 STL 顺序容器方法，string 还提供了大量有用的方法和友元（friend）函数。string 接口就是杂乱接口的一个很好的例子，这是第 5 章讨论过的一个设计陷阱。完整的 string 接口在本书网站上的标准库参数资源中做了总结；本节只是介绍了如何把 string 用作 STL 容器。

### 21.5.3 流作为 STL 容器

从传统意义上讲，输入和输出流并不是容器。它们不存储元素。不过，可以把流考虑成元素序列，因此与 STL 容器有一些共同的特性。C++ 流没有直接提供任何与 STL 相关的方法，但是 STL 提供了一些特殊的迭代器，名为 `istream_iterator` 和 `ostream_iterator`，由此可以“迭代”处理输入和输出流。第 23 章将解释如何使用这两个特殊的迭代器。

### 21.5.4 bitset

bitset 是位序列的一个定长抽象。应该记得，一位表示两个值，通常表示为 1 和 0、开和关、true 和 false。bitset 也使用术语设置（set）和反设置（unset）。可以对一位触发（toggle）或取反（flip），使之从一个值变为另一个值。

bitset 并不是真正的 STL 容器，它是定长的，并非对元素类型模板化，而且不支持迭代。不过，这是一个很有用的工具，经常与容器一同使用，所以在此对 bitset 做一个简要介绍。对 bitset 操作的完备总结请参考本书网站上的标准库参数资源。

#### bitset 基本知识

bitset 在 `<bitset>` 头文件中定义，它基于所存储的位数来实现模板化。默认构造函数会把 bitset 的所有位域初始化为 0。另一个构造函数可以从一个由 0 和 1 组成的 string 创建 bitset。

可以用 `set()`、`reset()` 和 `flip()` 方法调整单个位的值，而且可以用一个重载的 operator `[]` 访问和设置 bitset 的位域。需要注意，在一个非 const 对象上调用 operator `[]` 会返回一个代理对象，可以对其赋一个布尔值、调用 `flip()`，或者用 `~` 取反。还可以用 `test()` 方法访问单个的位域。

另外，可以用常规的流插入和析取操作符完成 bitset 的流输入和输出。bitset 会作为包括 0 和 1 的字符串流入流出。

以下是一个小例子。

```
#include <bitset>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
```

```
(
    bitset<10> myBitset;

    myBitset.set(3);
    myBitset.set(6);

    myBitset[8] = true;
    myBitset[9] = myBitset[3];

    if (myBitset.test(3)) {
        cout << "Bit 3 is set!\n";
    }
    cout << myBitset << endl;

    return (0);
)
```

输出为：

```
Bit 3 is set!
1101001000
```

要注意，输出 string 中最左字符是最高编号位（即编号或索引为 9）。

#### 位操作符

除了基本位处理例程，bitset 还提供了所有位操作符的实现： $\&$ 、 $|$ 、 $^$ 、 $\sim$ 、 $\ll$ 、 $\gg$ 、 $\&=$ 、 $|=$ 、 $^=$ 、 $\ll=$ 和 $\gg=$ 。这些操作符表现得就好像在“真的”位序列上操作一样。以下是一个例子。

```
#include <bitset>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    string str1 = "0011001100";
    string str2 = "0000111100";
    bitset<10> bitsOne(str1), bitsTwo(str2);

    bitset<10> bitsThree = bitsOne & bitsTwo;
    cout << bitsThree << endl;
    bitsThree <<= 4;
    cout << bitsThree << endl;

    return (0);
}
```

这个程序的输出为：

```
0000001100
0011000000
```

#### bitset 例子：表示电缆通道

bitset 的一种可能的使用是跟踪电缆订购用户所订购的通道。每个用户可能有一个与订购相关的通道 bitset，设置某些位表示他或她确实订购了这些通道。系统还支持通道“包”，也表示为 bitset，这是指通

常订购的通道组。

以下 CableCompany 类是这个模型的一个简单例子。它使用了两个 map，均为 string/bitset 映射，分别存储电缆包以及订购信息。

```
#include <bitset>
#include <map>
#include <string>
#include <stdexcept>
using std::map;
using std::bitset;
using std::string;
using std::out_of_range;

const int kNumChannels = 10;

class CableCompany
{
public:
    CableCompany() {}

    // Adds the package with the specified channels to the database
    void addPackage(const string& packageName,
                   const bitset<kNumChannels>& channels);

    // Removes the specified package from the database
    void removePackage(const string& packageName);

    // Adds the customer to the database with initial channels found in package
    // Throws out_of_range if the package name is invalid.
    void newCustomer(const string& name, const string& package)
        throw (out_of_range);

    // Adds the customer to the database with initial channels specified
    // in channels
    void newCustomer(const string& name, const bitset<kNumChannels>& channels);

    // Adds the channel to the customers profile
    void addChannel(const string& name, int channel);

    // Removes the channel from the customers profile
    void removeChannel(const string& name, int channel);

    // Adds the specified package to the customers profile
    void addPackageToCustomer(const string& name, const string& package);

    // Removes the specified customer from the database
    void deleteCustomer(const string& name);

    // Retrieves the channels to which this customer subscribes
    // Throws out_of_range if name is not a valid customer
    bitset<kNumChannels>& getCustomerChannels(const string& name)
        throw (out_of_range);

protected:
    typedef map<string, bitset<kNumChannels>> MapType;
    MapType mPackages, mCustomers;
};
```

以下是上述方法的实现。

```
#include "CableCompany.h"
using namespace std;

void CableCompany::addPackage(const string& packageName,
    const bitset<kNumChannels>& channels)
{
    // Just make a key/value pair and insert it into the packages map.
    mPackages.insert(make_pair(packageName, channels));
}

void CableCompany::removePackage(const string& packageName)
{
    // Just erase the package from the package map.
    mPackages.erase(packageName);
}

void CableCompany::newCustomer(const string& name, const string& package)
    throw (out_of_range)
{
    // Get a reference to the specified package.
    MapType::const_iterator it = mPackages.find(package);
    if (it == mPackages.end()) {
        // That package doesn't exist. Throw an exception.
        throw (out_of_range("Invalid package"));
    } else {
        // Create the account with the bitset representing that package.
        // Note that it refers to a name/bitset pair. The bitset is the
        // second field.
        mCustomers.insert(make_pair(name, it->second));
    }
}

void CableCompany::newCustomer(const string& name,
    const bitset<kNumChannels>& channels)
{
    // Just add the customer/channels pair to the customers map.
    mCustomers.insert(make_pair(name, channels));
}

void CableCompany::addChannel(const string& name, int channel)
{
    // Find a reference to the customers.
    MapType::iterator it = mCustomers.find(name);
    if (it != mCustomers.end()) {
        // We found this customer; set the channel.
        // Note that it is a reference to a name/bitset pair.
        // The bitset is the second field.
        it->second.set(channel);
    }
}

void CableCompany::removeChannel(const string& name, int channel)
{
    // Find a reference to the customers.
```



```

MapType::iterator it = mCustomers.find(name);
if (it != mCustomers.end()) {
    // We found this customer; remove the channel.
    // Note that it is a reference to a name/bitset pair.
    // The bitset is the second field.
    it->second.reset(channel);
}
}

void CableCompany::addPackageToCustomer(const string& name, const string& package)
{
    // Find the package.
    MapType::iterator itPack = mPackages.find(package);
    // Find the customer.
    MapType::iterator itCust = mCustomers.find(name);
    if (itCust != mCustomers.end() && itPack != mPackages.end()) {
        // Only if both package and customer are found, can we do the update.
        // Or-in the package to the customers existing channels.
        // Note that it is a reference to a name/bitset pair.
        // The bitset is the second field.
        itCust->second |= itPack->second;
    }
}

void CableCompany::deleteCustomer(const string& name)
{
    // Remove the customer with this name.
    mCustomers.erase(name);
}

bitset<kNumChannels>& CableCompany::getCustomerChannels(const string& name)
throw (out_of_range)
{
    // Find the customer.
    MapType::iterator it = mCustomers.find(name);
    if (it != mCustomers.end()) {
        // Found it!
        // Note that it is a reference to a name/bitset pair.
        // The bitset is the second field.
        return (it->second);
    }
    // Didn't find it. Throw an exception.
    throw (out_of_range("No customer of that name"));
}

```

最后，这个简单的程序展示了如何使用 CableCompany 类：

```

#include "CableCompany.h"
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    CableCompany myCC;

    string basic_pkg = "1111000000";
    string premium_pkg = "1111111111";
}

```



```
string sports_pkg = "0000100111";

myCC.addPackage("basic", bitset<kNumChannels>(basic_pkg));
myCC.addPackage("premium", bitset<kNumChannels>(premium_pkg));
myCC.addPackage("sports", bitset<kNumChannels>(sports_pkg));

myCC.newCustomer("Nicholas Solter", "basic");
myCC.addPackageToCustomer("Nicholas Solter", "sports");
cout << myCC.getCustomerChannels("Nicholas Solter") << endl;

return (0);
}
```

## 21.6 小结

本章介绍了标准模板库容器。另外还提供了许多示例代码来说明如何使用这些容器。希望你能充分领会 vector、deque、list、stack、queue、priority\_queue、map、multimap、set、multiset、string 和 bitset 的强大功能。即使你没有将其立即纳入到程序当中，至少要在心里留有它们的位置，以便在将来的项目中使用。

既然已经熟悉了容器，第 22 章将讨论通用算法，以展示 STL 的真正妙处。第 23 章是有关 STL 的第三部分，也是最后一部分，其中会介绍一些更为高级的特性，并提供一个示例容器和迭代器实现。

# 第五部分

## 使用库和模式

### 第 22 章 掌握 STL 算法和函数对象

第 21 章已经介绍了，STL 提供了大量通用数据结构。大多数其他的库都仅限于此，但 STL 则不同，它还包括另外的各种通用算法，除了少许例外，这些算法可以应用于任何容器的元素。通过使用这些算法，可以在容器中查找元素、对容器中的元素排序，处理容器中的元素，还能完成大量其他操作。算法之美就在于它们不仅独立于底层元素的类型，而且也独立于所操作容器的类型。算法仅使用迭代器接口来完成工作。

许多算法都接受回调（callback），这是一个函数指针，或者是一个类似函数指针的东西，如提供了重载 `operator()` 的对象。很方便地，STL 提供了一组类，可以用于为算法创建回调对象。这些回调对象称为函数对象（function object），或简称为 *functor*。

本章内容包括：

- 提供了算法的一个概述和 3 个示例算法：`find()`、`find_if()` 和 `accumulate()`
- 详细分析了函数对象
  - 介绍了预定义函数对象类：算术函数对象、比较函数对象和逻辑函数对象
  - 函数对象适配器
  - 如何编写自己的函数对象
- STL 算法详细内容
  - 工具算法
  - 非修改算法：查找、数值处理、比较和运算
  - 修改算法
  - 排序算法
  - 集合算法
- 一个大型例子：选民注册审计

#### 22.1 算法概述

算法背后的“神奇之处”在于：算法并非在容器本身运作，而是工作在迭代器这个“中间人”之上。采用这种方式，算法并没有绑定至特定的容器实现。所有 STL 算法都实现为函数模板，在此，模板类型参数通常是迭代器类型。迭代器本身指定为函数的实参。在第 11 章曾指出，模板化函数可以从函数参数推导出模板类型，所以一般可以像调用正常函数一样调用算法，而不用作为模板调用。

迭代器参数通常是迭代器区间。在第 21 章解释过，迭代器区间是半开区间，它包括区间中的首元素，但不包括末元素。这里的末尾迭代器实际上是一个“越过最后一个元素”的标记。

有些算法需要额外的模板类型参数和实参，有时这称为函数回调。这些回调可以是函数指针或函数对象。函数对象将在下一节详细讨论。首先，下面将仔细地分析一些算法。理解算法的最佳途径就是查看一些例子。看了这些算法如何工作后，其他算法的理解也就不难了。本节将详细介绍 `find()`、`find_if()` 和 `accumulate()` 算法。22.2 节介绍函数对象，22.3 节和 22.4 节讨论各类算法，并提供相应的示例。

### 22.1.1 `find()` 和 `find_if()` 算法

`find()` 在一个迭代器区间内查找一个特定元素。可以对任何类型容器的元素使用此算法。它会返回一个指示所找到元素的迭代器，或者是区间的末尾迭代器（未找到特定元素）。需要注意，`find()` 调用中指定的区间不必是容器中元素的整个区间，可以只是它的一个子集。

如果 `find()` 未能找到一个元素，它返回的迭代器等于函数调用中指定的末尾迭代器，而不是底层容器的末尾迭代器。

以下是 `find()` 的一个例子。

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    int num;

    vector<int> myVector;
    while (true) {
        cout << "Enter a number to add (0 to stop): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        myVector.push_back(num);
    }

    while (true) {
        cout << "Enter a number to lookup (0 to stop): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        vector<int>::iterator it = find(myVector.begin(), myVector.end(), num);
        if (it == myVector.end()) {
            cout << "Could not find " << num << endl;
        } else {
            cout << "Found " << *it << endl;
        }
    }

    return (0);
}
```

调用 `find()` 时以 `myVector.begin()` 和 `myVector.end()` 作为参数，以便查找 `vector` 的所有元素。以下是这个程序运行的情况：

```
Enter a number to add (0 to stop): 3
Enter a number to add (0 to stop): 4
Enter a number to add (0 to stop): 5
Enter a number to add (0 to stop): 6
Enter a number to add (0 to stop): 0
Enter a number to lookup (0 to stop): 5
Found 5
Enter a number to lookup (0 to stop): 8
Could not find 8
Enter a number to lookup (0 to stop): 4
Found 4
Enter a number to lookup (0 to stop): 2
Could not find 2
Enter a number to lookup (0 to stop): 0
```

有些容器，如 `map` 和 `set`，提供了自己的 `find()` 版本作为类方法。

如果容器提供了一个方法，其功能与某个通用算法相同，应当使用这个方法才对，因为容器提供的方法更快一些。例如，通用 `find()` 算法会在线性时间内运行，即使是利用一个 `map` 迭代器也不例外，而映射所提供的 `find()` 方法则在对数时间内完成。

`find_if()` 类似于 `find()`，只不过它接受一些谓词函数回调（predicate function callback）而不是一个待匹配的元素。谓词会返回 `true` 或 `false`。`find_if()` 在区间中每个元素上调用此谓词，直到谓词返回 `true`。此时，`find_if()` 会返回一个指示该元素的迭代器。下面的程序从用户那里读取测验分数，然后检查分数是否“完美”。完美的分数应该等于 100 或者更高。这个程序与前面的程序很类似，在此突出显示了二者的区别。

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;
```

```
bool perfectScore(int num)
{
    return (num >= 100);
}
```

```
int main(int argc, char** argv)
{
    int num;

    vector<int> myVector;
    while (true) {
        cout << "Enter a test score to add (0 to stop): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        myVector.push_back(num);
    }
}
```

```
vector<int>::iterator it = find_if(myVector.begin(), myVector.end(),
    perfectScore);
if (it == myVector.end()) {
    cout << "No perfect scores\n";
} else {
    cout << "Found a \"perfect\" score of " << *it << endl;
}
return (0);
}
```

这个程序向 `find_if()` 算法传递 `perfectScore()` 函数的一个指针，`find_if()` 算法再对每个元素调用此函数，直到返回 `true` 为止。

遗憾的是，STL 没有提供 `find_all()` 或相应的算法来返回与一个谓词匹配的所有实例。第 23 章将介绍如何编写自己的 `find_all()` 算法。

### 22.1.2 accumulate() 算法

计算容器中所有元素的总和或者其他某个算术量往往很有用。`accumulate()` 函数就是做这个工作的。采用其最基本的形式，它会计算给定区间内所有元素的总和。例如，以下函数会计算 `vector` 中一个整数序列的算术平均值。算术平均值就是所有元素之和除以元素个数。

```
#include <numeric>
#include <vector>
using namespace std;

double arithmeticMean(const vector<int>& nums)
{
    double sum = accumulate(nums.begin(), nums.end(), 0);
    return (sum / nums.size());
}
```

注意，`accumulate()` 在 `<numeric>` 中声明，而不是 `<algorithm>`。还要注意，`accumulate()` 将第三个参数作为总和的初始值，这里应当为 0（加法单元或幺元），以便开始一次新的求和。

第二种形式的 `accumulate()` 允许调用者指定要完成的操作（而不是加法）。这个操作采取二元回调的方式。假想计算几何平均值，即以序列中所有数字之积为底，以序列大小的倒数为指数计算幂值。在这种情况下，你可能想使用 `accumulate()` 来计算乘积而不是求和。可以编写如下的代码：

```
#include <numeric>
#include <vector>
#include <cmath>
using namespace std;

int product(int num1, int num2)
{
    return (num1 * num2);
}

double geometricMean(const vector<int>& nums)
{
    double mult = accumulate(nums.begin(), nums.end(), 1, product);
    return (pow(mult, 1.0 / nums.size()));
}
```

要注意, `product()` 函数作为一个回调传递给 `accumulate()`, 此聚集的初始值为 1 (乘法单元或幺元) 而不是 0。22.2 节将介绍如何在 `geometricMean()` 函数中使用 `accumulate()`, 而无需编写一个函数回调。

## 22.2 函数对象

既然已经了解了一些 STL 算法。应该可以领会函数对象了。在第 16 章曾指出, 可以在一个类中重载函数调用操作符, 使得该类的对象可以用于替代函数指针。这些对象称为函数对象, 或简称 functor。

许多 STL 算法, 如 `find_if()` 和第二种形式的 `accumulate()`, 都需要一个函数指针作为参数之一。在使用这些函数时, 可以传递一个函数对象而不是函数指针。这一点本身并不是我们踊跃采用函数对象的根本原因。你当然可以编写自己的函数对象类, 但函数对象真正的魅力在于, C++ 提供了许多预定义的函数对象类, 它们可以完成最常用的回调操作。本节将介绍这些预定义的类, 并说明如何加以使用。

所有预定义的函数对象类都位于 `<functional>` 头文件中。

### 22.2.1 算术函数对象

C++ 为 5 个二元算术操作符提供了函数对象类模板: `plus`、`minus`、`multiplies`、`divides` 和 `modulus`。另外, 还提供了一元的 `negate`。这些类都针对操作数类型进行了模板化, 而且是具体操作符的包装器。它们取一个或两个模板类型参数, 完成操作, 并返回结果, 以下是一个使用 `plus` 类模板的例子。

```
#include <functional>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    plus<int> myPlus;

    int res = myPlus(4, 5);
    cout << res << endl;

    return (0);
}
```

这个例子很傻, 因为完全可以直接使用 `operator+`, 为什么还要用 `plus` 类模板呢? 算术函数对象的好处在于, 可以将其作为回调传递给算法, 如果利用算术操作符则无法直接做到这一点。

例如, 本章较早前 `geometricMean()` 函数的实现使用了 `accumulate()` 函数, 并提供了一个函数指针回调来完成两个整数的相乘。可以对其重写, 转而使用 `multiplies` 函数对象:

```
#include <numeric>
#include <vector>
#include <cmath>
#include <functional>
using namespace std;

double geometricMean(const vector<int>& nums)
{
    double mult = accumulate(nums.begin(), nums.end(), 1,
        multiplies<int>());
    return (pow(mult, 1.0 / nums.size()));
}
```



表达式 `multiplies<int>()` 会创建 `multiplies` 类的一个新对象，并基于 `int` 类型对其实例化。其他算术函数对象也有类似的行为。

算术函数对象只是包装在算术操作符之外的包装器。如果把函数对象用作算法中的回调，要保证容器中的对象实现了适当的操作，如 `operator*` 或 `operator+`。

### 22.2.2 比较函数对象

除了算术函数对象类，C++ 语言还提供了所有标准比较：`equal_to`、`not_equal_to`、`less`、`greater`、`less_equal` 和 `greater_equal`。在第 21 章已经见过 `less`，这是 `priority_queue` 和关联容器中元素采用的默认比较。现在可以看一看如何改变这个规则，以下是一个使用默认比较操作符 `less` 的 `priority_queue` 例子。

```
#include <queue>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    priority_queue<int> myQueue;

    myQueue.push(3);
    myQueue.push(4);
    myQueue.push(2);
    myQueue.push(1);

    while (!myQueue.empty()) {
        cout << myQueue.top() << endl;
        myQueue.pop();
    }

    return (0);
}
```

程序的输出如下所示：

```
4
3
2
1
```

可以看到，按照 `less` 比较，`queue` 的元素以降序删除，可以把比较修改为 `greater`，只需将其指定为比较模板参数即可。第 21 章中 `priority_queue` 模板定义如下所示：

```
template <typename T, typename Container = vector<T>, typename Compare = less<T>> ;
```

遗憾的是，`Compare` 类型参数位于最后面，这说明，要想指定比较，还必须指定容器。以下例子对上述程序做了修改，以便 `priority_queue` 使用 `greater` 按升序对元素排序：

```
#include <queue>
#include <functional>
#include <iostream>
using namespace std;
```

```
int main(int argc, char** argv)
{
    priority_queue<int, vector<int>, greater<int> > myQueue;

    myQueue.push(3);
    myQueue.push(4);
    myQueue.push(2);
    myQueue.push(1);

    while (!myQueue.empty()) {
        cout << myQueue.top() << endl;
        myQueue.pop();
    }

    return (0);
}
```

输出现在如下所示：

```
1
2
3
4
```

本章后面要学到的许多算法都需要比较回调，对此，预定义比较函数对象会很方便。

### 22.2.3 逻辑函数对象

C++ 还为 3 个逻辑操作提供了函数对象类：logical\_not、logical\_and 和 logical\_or。不过，它们对于标准 STL 通常用处不大。

### 22.2.4 函数对象适配器

想使用标准提供的基本函数对象时，通常就感觉好像要把一个方桩子塞进一个圆洞里一样，总有些不对劲。例如，不能对 find\_if() 使用基本比较函数对象，因为 find\_if() 每次只向回调传递一个参数而不是两个。函数适配器（function adapter）就是为了矫正这样一些问题。它们为函数组合（functional composition）提供了一些支持，即把函数组合在一起来创建所需的行为。

#### 捆绑器

假设你想使用 find\_if() 算法找到序列中第一个大于或等于 100 的元素。为了解决本章前面讨论的问题，我们编写了一个函数 perfectScore()，并将函数指针传递给 find\_if()。你已经了解了比较函数对象，看上去好像能够使用 greater\_equal 类模板来实现一个解决方案。

greater\_equal 类模板的问题是它取两个参数，而 find\_if() 每次只会为回调谓词传递一个参数。应当能够指定 find\_if() 要使用 greater\_equal，但是每次都要把 100 作为第二个实参传入。这样一来，序列中的每个元素都会与 100 比较。幸运的是，C++ 恰好为此提供了一种方法：

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
using namespace std;
```

```

int main(int argc, char** argv)
{
    int num;

    vector<int> myVector;
    while (true) {
        cout << "Enter a test score to add (0 to stop): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        myVector.push_back(num);
    }

    vector<int>::iterator it = find_if(myVector.begin(), myVector.end(),
        bind2nd(greater_equal<int>(), 100));
    if (it == myVector.end()) {
        cout << "No perfect scores\n";
    } else {
        cout << "Found a \"perfect\" score of " << *it << endl;
    }
    return (0);
}

```

`bind2nd()` 函数称为一个捆绑器 (binder)，因为它把值 100 作为第二个参数捆绑至 `greater_equal`。结果是 `find_if()` 会用 `greater_equal` 来对每个元素与 100 进行比较。

可以对任何二元函数使用 `bind2nd()`。另外还有一个与之对应的 `bind1st()` 函数，它能把一个实参绑定为一个二元函数的第一个参数。

### 取反器

取反器 (negator) 函数与捆绑器类似，只是对一个谓词的结果取反。例如，如果想找出序列中第一个小于 100 的元素，可以对 `greater_equal` 的结果应用一个取反器适配器，如下所示：

```

int main(int argc, char** argv)
{
    int num;

    vector<int> myVector;
    while (true) {
        cout << "Enter a test score to add (0 to stop): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        myVector.push_back(num);
    }

    vector<int>::iterator it = find_if(myVector.begin(), myVector.end(),
        not1(bind2nd(greater_equal<int>(), 100)));
    if (it == myVector.end()) {
        cout << "All perfect scores\n";
    } else {
        cout << "Found a \"less-than-perfect\" score of " << *it << endl;
    }
    return (0);
}

```

函数 `not1()` 以每个谓词调用的结果作为参数，并对其取反。当然，还可以直接使用 `less` 而不是 `greater_equal`。有一些情况下，通常在使用非标准的函数对象时，`not1()` 会很方便。`not1()` 中的 “1” 是指其操作数必须是一个一元函数（要取一个参数）。如果操作数是一个二元函数（取两个参数），就必须使用 `not2()`。需要注意，在这里要使用 `not1()`，尽管 `greater_equal` 是一个二元函数，因为 `bind2nd()` 已经将第二个实参总是绑定为 100，因此已经把二元函数转换成一个一元函数了。

可以看到，使用函数对象和适配器可能会很快变得很复杂。我们的建议是：在目的很清楚、很好理解的简单情况下限制其使用，但是对于更为复杂的情况，可以编写自己的函数对象或采用显式的循环。

### 调用成员函数

如果你有一个包括对象的容器，有时可能想把一个类方法的指针作为回调传递给一个算法。例如，可能想对序列中的每个 `string` 调用 `empty()`，从而找到包含 `string` 的 `vector` 中的第一个空 `string`。不过，如果只是向 `find_if()` 传递 `string::empty()` 的一个指针，算法就没有办法知道它接收的是一个方法的指针，而不是一个正常的函数指针或函数对象。第 9 章曾经解释过，调用一个方法指针的代码与调用常规函数指针的代码有所不同，因为前者必须在一个对象的上下文中调用。因此，C++ 提供了一个名为 `mem_fun_ref()` 的转换函数，在把一个方法指针传递给算法时，可以先对其调用此转换函数（`mem_fun_ref()` 中的 “fun” 是指 “函数”（function），完全不是有趣（fun）的意思）。可以如下使用：

```
#include <functional>
#include <algorithm>
#include <string>
#include <vector>
#include <iostream>
using namespace std;

void findEmptyString(const vector<string>& strings)
{
    vector<string>::const_iterator it = find_if(strings.begin(), strings.end(),
        mem_fun_ref(&string::empty));

    if (it == strings.end()) {
        cout << "No empty strings!\n";
    } else {
        cout << "Empty string at position: " << it - strings.begin() << endl;
    }
}
```

`mem_fun_ref()` 生成一个函数对象，它会作为 `find_if()` 的回调。每次回调时，它都会对参数调用 `empty()` 方法。

`mem_fun_ref()` 也可用于 0 参数方法和一元方法。在希望有一个一元或二元函数的位置上，可以分别将 `mem_fun_ref()` 作用于 0 参数方法和一元方法的结果用作回调。

如果容器中是对象的指针而不是对象本身，必须使用一个不同的适配器 `mem_fun()` 来调用成员函数。例如：

```
#include <functional>
#include <algorithm>
#include <string>
#include <vector>
#include <iostream>
using namespace std;
```

```

void findEmptyString(const vector<string*>& strings)
{
    vector<string*>::const_iterator it = find_if(strings.begin(), strings.end(),
        mem_fun(&string::empty));

    if (it == strings.end()) {
        cout << "No empty strings!\n";
    } else {
        cout << "Empty string at position: " << it - strings.begin() << endl;
    }
}

```

### 适配实际函数

不能对函数适配器 `bind1st()`、`bind2nd()`、`not1()` 或 `not2()` 直接使用常规的函数指针，因为这些适配器需要其适配函数对象的特定 `typedef`。因此，C++ 标准库提供的最后一个函数适配器 `ptr_fun()` 允许将常规的函数指针以某种形式包装，以便用于适配器。这主要用于使用传统的 C 函数，如 C 标准库中的函数。如果你编写了自己的回调，建议你编写函数对象类，22.2.5 节将做相关说明。

例如，假设你想编写一个函数 `isNumber()`，如果 `string` 中的每个字符都是一个数字字符，则返回 `true`。在第 21 章中介绍过，C++ `string` 提供了一个迭代器。因此，可以使用 `find_if()` 算法来搜索字符串中第一个非数字字符。如果找到了这样一个字符，这个字符串就不是一个数字。`<cctype>` 头文件提供了一个传统 C 函数，名为 `isdigit()`，如果字符是数字，它会返回 `true`，否则返回 `false`。问题在于，你想找第一个不是数字的字符，这需要 `not1()` 适配器。不过，由于 `isdigit()` 是一个 C 函数，而不是一个函数对象，因此需要使用 `ptr_fun()` 适配器生成一个可以用于 `not1()` 的函数对象。代码如下：

```

#include <functional>
#include <algorithm>
#include <cctype>
#include <string>
using namespace std;

bool isNumber(const string& str)
{
    string::const_iterator it = find_if(str.begin(), str.end(),
        not1(ptr_fun(&isdigit)));
    return (it == str.end());
}

```

注意这里使用了 `::` 作用域解析操作符来指定应当在全局作用域中查找 `isdigit()`。

### 22.2.5 编写自己的函数对象

当然，你也可以编写自己的函数对象来完成更特定的任务，而不只是预定义函数对象所提供的功能。如果想要对这些函数对象使用函数适配器，必须提供某种 `typedef`。为此，最简单的办法就是从 `unary_function` 或 `binary_function` 派生你的函数对象类，究竟由哪个类派生取决于取一个还是两个参数。这两个类都在 `<functional>` 中定义，它们针对所提供“函数”的参数和返回类型进行模板化。例如，可以不使用 `ptr_fun()` 来转换 `isdigit()`，而是编写如下的一个包装器函数对象：

```

#include <functional>
#include <algorithm>
#include <cctype>
#include <string>
using namespace std;

class myIsDigit : public unary_function<char, bool>
{
public:
    bool operator() (char c) const { return (::isdigit(c)); }
};

bool isNumber(const string& str)
{
    string::const_iterator it = find_if(str.begin(), str.end(),
        not1(myIsDigit()));
    return (it == str.end());
}

```

注意，myIsDigit 类的重载函数调用操作符必须是 const，以便为 find\_if() 传递对象。

算法可以建立函数对象谓词的多个副本，并对不同的元素调用不同的副本。因此，编写算法时，不能指望调用之间对象的内部状态会保持一致。

## 22.3 算法细节

本章介绍了几类通用算法，并提供了各类算法的例子。本书网站上的标准库参数资源对所有算法提供了一个总结，不过有关的详细内容，还是应当参考附录 B 中所列的有关 STL 的参考书。

第 21 章曾指出，共有 5 类迭代器：输入、输出、前向、双向和随机访问。这些迭代器没有正式的类层次体系，因为各个容器的实现并非标准体系中的一部分。不过，基于这些迭代器所要提供的功能，可以推导出一个层次体系来。具体地，每个随机访问迭代器也是双向迭代器，每个双向迭代器也是前向迭代器，而每个前向迭代器也还是输入和输出迭代器。

算法要指定需要何种迭代器，一种标准方法就是使用以下名字作为迭代器模板参数：InputIterator，OutputIterator，ForwardIterator，BidirectionalIterator 和 RandomAccessIterator。这些名字仅仅是名字而已，它们不提供绑定类型检查。因此，例如，调用一个需要有 RandomAccessIterator 的算法时，为之传递一个双向迭代器是允许的。模板并不会做类型检查，因此它允许这种实例化。不过，函数中使用随机访问迭代器功能的代码针对双向迭代器则无法编译通过。因此，这个需求是有的，但是与想像中施加限制的位置不一样（译者注：即你以为在传递不同类型迭代器时就应该做出限制，但是实际上只有存在具体的迭代器调用时才会出问题）。因此错误消息就有些让人混淆。例如，如果想使用通用 sort() 算法，它需要 list 上的一个随机访问迭代器，但只提供了一个双向迭代器，在 g++ 中会给出以下错误：

```

/usr/include/c++/3.2.2/bits/stl_algo.h: In function 'void
std::sort(_RandomAccessIter, _RandomAccessIter) [with _RandomAccessIter =
std::_List_iterator<int, int&, int*>]':
Sorting.cpp:38: instantiated from here
/usr/include/c++/3.2.2/bits/stl_algo.h:2178: no match for `
std::_List_iterator<int, int&, int*>& - std::_List_iterator<int, int&,
int*>&' operator

```



如果你还不懂这个错误，也不用担心。sort()算法将在本章后面介绍。

大多数算法都在<algorithm>头文件中定义，不过有些算法放在<numeric>中。他们都在std命名空间中。有关的详细内容请参见本书网站上的标准库参数资源。

### 22.3.1 工具算法

STL 提供了三个实现为模板的工具算法：min()、max()和swap()。min()和max()用operator< 或一个用户提供的二元谓词对任意类型的两个元素进行比较，分别返回较小或较大元素的一个引用。swap()按引用取任意类型的两个元素，并交换其值。

这些工具并不作用于元素序列，因此不取迭代器参数。

以下程序展示了这三个函数：

```
#include <algorithm>
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    int x = 4, y = 5;
    cout << "x is " << x << " and y is " << y << endl;
    cout << "Max is " << max(x, y) << endl;
    cout << "Min is " << min(x, y) << endl;
    swap(x, y);
    cout << "x is " << x << " and y is " << y << endl;
    cout << "Max is " << max(x, y) << endl;
    cout << "Min is " << min(x, y) << endl;

    return (0);
}
```

程序的输出如下：

```
x is 4 and y is 5
Max is 5
Min is 4
x is 5 and y is 4
Max is 5
Min is 4
```

### 22.3.2 非修改算法

非修改算法包括以下函数：搜索区间中的元素、生成区间中元素的相关数值信息、将两个区间相互比较，以及处理区间中的各个元素。

#### 搜索算法

你已经见到了两个搜索算法的例子：find()和find\_if()。STL 提供了基本 find()算法的几个其他版本，可以作用于未排序的元素序列。adjacent\_find()会查找彼此相等的两个连续元素的第一个实例。find\_first\_of()同时查找多个值中的某一个。search()和find\_end()查找与一个指定元素序列匹配的子序列，分别从给定区间的开始和结束处开始查找。search\_n()可以认为是search()的一个特例，或者是adjacent\_find()的一般情况，它会查找第一个包括n个连续元素的序列，这些元素都与一个给定值匹配。最后，min\_element()和max\_element()可以找出序列中最小

或最大元素。

`find_end()` 与 `search()` 对应，只不过是序列最后开始查找，而不是从起始处开始查找。它并不是 `find()` 的逆算法。`find()`、`find_if()` 或其他搜索单元素的算法没有逆算法，因为可以使用一个 `reverse_iterator` 达到同样的效果。`reverse_iterators` 将在第 23 章介绍。

`find()`、`adjacent_find()`、`min_element()` 和 `max_element()` 都在线性时间内运行。其他算法的运行时间则为二次方时间（译者注： $O(n^2)$ ）。所有算法都使用默认的 `operator==` 或 `operator<` 比较，不过也提供了重载的版本，允许客户指定另外的一个比较回调。

以下是前述搜索算法的一些例子。

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main(int argc, char** argv)
{
    // The list of elements to be searched
    int elems[] = {5, 6, 9, 8, 8, 3};
    // Construct a vector from the list, exploiting the
    // fact that pointers are iterators too.
    vector<int> myVector(elems, elems + 6);
    vector<int>::const_iterator it, it2;

    // Find the min and max elements in the vector.
    it = min_element(myVector.begin(), myVector.end());
    it2 = max_element(myVector.begin(), myVector.end());
    cout << "The min is " << *it << " and the max is " << *it2 << endl;

    // Find the first pair of matching consecutive elements.
    it = adjacent_find(myVector.begin(), myVector.end());
    if (it != myVector.end()) {
        cout << "Found two consecutive equal elements of value "
             << *it << endl;
    }

    // Find the first of two values.
    int targets[] = {8, 9};
    it = find_first_of(myVector.begin(), myVector.end(), targets,
                      targets + 2);

    if (it != myVector.end()) {
        cout << "Found one of 8 or 9: " << *it << endl;
    }

    // Find the first subsequence.
    int sub[] = {8, 3};
    it = search(myVector.begin(), myVector.end(), sub, sub + 2);
    if (it != myVector.end()) {
        cout << "Found subsequence 8, 3 at position " << it - myVector.begin()
             << endl;
    }

    // Find the last subsequence (which should be the same as the first).
    it2 = find_end(myVector.begin(), myVector.end(), sub, sub + 2);
```

```
if (it != it2) {
    cout << "Error: search and find_end found different subsequences "
        << " even though there is only one match.\n";
}

// Find the first subsequence of two consecutive 8s.
it = search_n(myVector.begin(), myVector.end(), 2, 8);
if (it != myVector.end()) {
    cout << "Found two consecutive 8s starting at position "
        << it - myVector.begin() << endl;
}

return (0);
}
```

输出如下：

```
The min is 3 and the max is 9
Found two consecutive equal elements of value 8
Found one of 8 or 9: 9
Found subsequence 8, 3 at position 4
Found two consecutive 8s starting at position 3
```

还有一些搜索算法仅作用于有序序列：binary\_search()、lower\_bound()、upper\_bound()和equal\_range()。binary\_search()会在对数时间找到一个匹配元素。其他三个算法与map和set容器上的相应方法类似。有关内容请参见第21章和本书网站上的标准库参数资源。

要记住，如果有可用的相应容器方法，就应当使用这些方法，而不要使用算法，因为方法的效率更高。

### 数值处理算法

你已经看到一个数值处理算法的例子了：accumulate()。另外，count()和count\_if()算法可以用于计算容器中有给定值的元素的个数。其作用类似于map和set容器上的count()方法。

其他数值处理算法的用处就不太大了，所以在此不再讨论。如果感兴趣，可以参见本书网站上的标准库参数资源。

### 比较算法

可以采用三种不同方法来比较整个元素区间：equal()、mismatch()和lexicographical\_compare()。这些算法都会按顺序比较两个区间中相应位置上的元素。如果所有对应元素都相等，则equal()返回true。mismatch()返回的迭代器会指示各区间中对应元素不相等的第一个位置。如果第一个区间中所有元素都小于第二个区间中相应元素，或者如果第一个区间比第二个区间短，而且到第一个区间结束之前的所有元素都比第二个区间中的相应元素小，lexicographical\_compare()就返回true。可以把这个函数认为是非字符元素字母化的一个一般化（泛化）。

如果想比较两个相同类型容器中的元素，可以使用operator==或operator<，而不用采用equal()或lexicographical\_compare()。算法主要用于比较不同类型容器中的元素序列。

以下是equal()、mismatch()和lexicographical\_compare()的一些例子。

```
#include <algorithm>
#include <vector>
#include <list>
#include <iostream>
using namespace std;

// Function template to populate a container of ints.
// The container must support push_back().
template<typename Container>
void populateContainer(Container& cont)
{
    int num;

    while (true) {
        cout << "Enter a number (0 to quit): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        cont.push_back(num);
    }
}

int main(int argc, char** argv)
{
    vector<int> myVector;
    list<int> myList;

    cout << "Populate the vector:\n";
    populateContainer(myVector);
    cout << "Populate the list:\n";
    populateContainer(myList);
    if (myList.size() < myVector.size()) {
        cout << "Sorry, the list is not long enough.\n";
        return (0);
    }

    // Compare the two containers.
    if (equal(myVector.begin(), myVector.end(), myList.begin())) {
        cout << "The two containers have equal elements\n";
    } else {
        // If the containers were not equal, find out why not.
        pair<vector<int>::iterator, list<int>::iterator> miss =
            mismatch(myVector.begin(), myVector.end(), myList.begin());
        cout << "The first mismatch is at position "
            << miss.first - myVector.begin() << ". The vector has value "
            << *(miss.first) << " and the list has value " << *(miss.second)
            << endl;
    }

    // Now order them.
    if (lexicographical_compare(myVector.begin(), myVector.end(), myList.begin(),
        myList.end())) {

```

```

        cout << "The vector is lexicographically first.\n";
    } else {
        cout << "The list is lexicographically first.\n";
    }

    return (0);
}

```

这个程序的运行情况如下：

```

Populate the vector:
Enter a number (0 to quit): 5
Enter a number (0 to quit): 6
Enter a number (0 to quit): 7
Enter a number (0 to quit): 8
Enter a number (0 to quit): 0
Populate the list:
Enter a number (0 to quit): 5
Enter a number (0 to quit): 6
Enter a number (0 to quit): 7
Enter a number (0 to quit): 9
Enter a number (0 to quit): 0
The first mismatch is at position 3. The vector has value 8 and the list has value 9
The vector is lexicographically first.

```

### 运算算法

这一类中只有一个算法：`for_each()`。不过，这也是STL中最有用的算法之一。它会对区间中的各个元素执行一个回调。可以利用简单的函数回调来完成一些简单工作，如打印出容器中的每个元素。例如：

```

#include <algorithm>
#include <map>
#include <iostream>
using namespace std;

void printPair(const pair<int, int>& elem)
{
    cout << elem.first << "->" << elem.second << endl;
}

int main(int argc, char** argv)
{
    map<int, int> myMap;
    myMap.insert(make_pair(4, 40));
    myMap.insert(make_pair(5, 50));
    myMap.insert(make_pair(6, 60));
    myMap.insert(make_pair(7, 70));
    myMap.insert(make_pair(8, 80));

    for_each(myMap.begin(), myMap.end(), &printPair);

    return (0);
}

```

还可以使用一个函数对象保留元素之间的信息，来完成更难的任务。`for_each()`会返回回调对象的一个副本，所以可以在函数对象中积累信息，`for_each()`完成对各个元素的处理后，可以从该函数对象获取



所积累的信息。例如，可以编写一个函数对象同时记录到目前为止的最小元素和最大元素，这样只需一趟就可以同时计算出 min 和 max 元素。以下例子中所示的 MinAndMax 函数对象假设调用此函数对象的区间中至少包括一个元素。它使用一个布尔 first 变量将 min 和 max 初始化为第一个元素，在此之后，它将每个后续的元素与当前存储的 min 和 max 值进行比较。

```
#include <algorithm>
#include <functional>
#include <vector>
#include <iostream>
using namespace std;

// The populateContainer() function is identical to the one shown above for
// comparison algorithms, so is omitted here.

class MinAndMax : public unary_function<int, void>
{
public:
    MinAndMax();
    void operator()(int elem);

    // Make min and max public for easy access.
    int min, max;

protected:
    bool first;
};

MinAndMax::MinAndMax() : min(-1), max(-1), first(true)
{
}

void MinAndMax::operator()(int elem)
{
    if (first) {
        min = max = elem;
    } else if (elem < min) {
        min = elem;
    } else if (elem > max) {
        max = elem;
    }
    first = false;
}

int main(int argc, char** argv)
{
    vector<int> myVector;
    populateContainer(myVector);
    MinAndMax func;
    func = for_each(myVector.begin(), myVector.end(), func);
    cout << "The max is " << func.max << endl;
    cout << "The min is " << func.min << endl;

    return (0);
}
```



你可能想忽略 `for_each()` 的返回值，不过还是应该在调用之后从 `func` 读取信息。不过，在此这是无法做到的，因为 `func` 不一定按引用传入 `for_each()`。必须保留返回值，以确保行为正确。

关于 `for_each()` 还有最后一点，回调可以按引用取参数，而且可以修改参数。其效果就是可以修改实际迭代器区间中的值。本章后面的选民注册示例展示了这个功能的使用。

### 22.3.3 修改算法

STL 提供了大量修改算法，可以完成许多任务，如从一个区间将元素复制到另一个区间，删除元素，或把区间中的元素逆置。

修改算法都有源 (source) 和目标 (destination) 区间的概念。元素从源区间读取，增加到或者经修改置于目标区间中。源和目标区间通常可以是一样的，在这种情况下，算法称为就地 (in place) 操作。

`map` 和 `multimap` 的区间不能用作修改算法的目标。这些算法会覆盖完整的元素，`map` 中包含的元素是键/值对。不过，`map` 和 `multimap` 都把键标记为 `const`，所以不能对其赋值。类似地，许多 `set` 和 `multiset` 的实现只对元素提供 `const` 迭代，所以也不能使用这些容器的区间作为修改算法的目标。替代做法是使用一个插入迭代器 (insert iterator)，有关内容见第 23 章的介绍。

#### 转换

`transform()` 算法类似于 `for_each()`，其中为区间中的每个元素提供了一个回调。区别在于 `transform()` 希望回调为每个调用生成一个新元素，它会把生成的元素保存在指定的目标区间中。如果想通过完成转换，将一个区间中的各个元素代之以调用回调后得到的结果，那么源和目标区间可以是一样的。例如，可以向一个 `vector` 中的各个元素增加 100，如下所示：

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
using namespace std;

// The populateContainer() function is identical to the one shown above for
// comparison algorithms, so is omitted here.

void print(int elem)
{
    cout << elem << " ";
}

int main(int argc, char** argv)
{
    vector<int> myVector;
    populateContainer(myVector);
    cout << "The vector contents are:\n";
    for_each(myVector.begin(), myVector.end(), &print);
    cout << endl;
    transform(myVector.begin(), myVector.end(), myVector.begin(),
        bind2nd(plus<int>(), 100));
    cout << "The vector contents are:\n";
    for_each(myVector.begin(), myVector.end(), &print);
    cout << endl;
    return (0);
}
```

另一种形式的 `transform()` 会对区间中的元素对调用一个二元函数。有关内容请参见本书网站上的标准库参数资源。有意思的是，通过为 `transform()` 编写合适的函数对象，就能利用它得到其他许多修改算法的功能，如 `copy()` 和 `replace()`。不过，在可能的情况下使用更简单的算法还是会方便一些。

`transform()` 和其他修改算法通常会返回一个指示目标区间“越过最后一个元素”值的迭代器。本书中的例子一般会忽略这个返回值，有关内容请参见本书网站上的标准库参数资源。

### 复制

`copy()` 算法允许你将元素从一个区间复制到另一个区间。源和目标区间必须不同，但是它们可以交叠。要注意，`copy()` 并不会把元素插入到目标区间。它只是对已经有的元素进行重写（覆盖）。因此，不能直接使用 `copy()` 把元素插入到一个容器中，只能用它来覆盖容器中已经有的元素。

第 23 章介绍了如何使用迭代器适配器利用 `copy()` 将元素插入到一个容器或流中。

以下是一个 `copy()` 的例子，这里在 `vector` 上利用 `resize()` 方法以确保目标容器中有足够的空间。

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

// The populateContainer() and print() functions are identical to those
// in the previous example, so are omitted here.

int main(int argc, char** argv)
{
    vector<int> vectOne, vectTwo;
    populateContainer(vectOne);

    vectTwo.resize(vectOne.size());
    copy(vectOne.begin(), vectOne.end(), vectTwo.begin());
    for_each(vectTwo.begin(), vectTwo.end(), &print);

    return (0);
}
```

### 替换

`replace()` 和 `replace_if()` 算法会把一个区间中分别与某个值或某个谓词匹配的元素替换为一个新值。例如，可以要求一个整数区间中的所有元素都介于 0~100 之间，为此要把所有小于 0 的值替换为 0，所有大于 100 的值替换为 100；

```
#include <algorithm>
#include <functional>
#include <vector>
#include <iostream>
using namespace std;

// The populateContainer() and print() functions are identical to those
// in the previous example, so are omitted here.

int main(int argc, char** argv)
```

```
{
    vector<int> myVector;
    populateContainer(myVector);
    replace_if(myVector.begin(), myVector.end(), bind2nd(less<int>(), 0), 0);
    replace_if(myVector.begin(), myVector.end(), bind2nd(greater<int>(), 100),
        100);
    for_each(myVector.begin(), myVector.end(), &print);
    cout << endl;

    return (0);
}
```

replace()还有一些变种, 名为 replace\_copy()和 replace\_copy\_if(), 它们会把结果复制到不同的目标区间。

### 删除

remove() 和 remove\_if() 算法会从一个区间删除某些元素。要删除的元素可以用一个特定值来指定, 也可以用谓词来指定。要记住重要的一点, 这些元素不会从底层容器中删除, 因为算法只能访问迭代器抽象, 而不是访问容器。相反, 所删除的元素会复制到区间的最后, 并返回新的区间末尾 (比原来要短)。如果确实想从容器清除所删除的元素, 就必须使用 remove() 算法, 然后在容器上调用 erase()。

以下是一个函数例子, 它从包括 string 的 vector 中删除空的 string。这个函数类似于本章前面给出的函数 findEmptyString()。

```
#include <functional>
#include <algorithm>
#include <string>
#include <vector>
#include <iostream>
using namespace std;

void removeEmptyStrings(vector<string>& strings)
{
    vector<string>::iterator it = remove_if(strings.begin(), strings.end(),
        mem_fun_ref(&string::empty));
    // Erase the removed elements.
    strings.erase(it, strings.end());
}

void printString(const string& str)
{
    cout << str << " ";
}

int main(int argc, char** argv)
{
    vector<string> myVector;
    myVector.push_back("");
    myVector.push_back("stringone");
    myVector.push_back("");
    myVector.push_back("stringtwo");
    myVector.push_back("stringthree");
    myVector.push_back("stringfour");

    removeEmptyStrings(myVector);
```

```
cout << "Size is " << myVector.size() << endl;
for_each(myVector.begin(), myVector.end(), &printString);
cout << endl;
return (0);
}
```

`remove()` 有两个变种：`remove_copy()` 和 `remove_copy_if()`，它们不会修改源区间。相反，会把所有未删除的元素复制到一个不同的目标区间。这些算法与 `copy()` 很类似，因为区间必须足够大来容纳新的元素。

`remove()` 系列函数是稳定的 (stable)，因为即使把所删除元素移到最后，容器中剩下的元素顺序依然保持。

### 惟一

`unique()` 算法是 `remove()` 的一个特例，会删除所有重复的连续元素。在第 21 章曾提到，`list` 容器提供了一个 `unique()` 方法，也可以提供同样的语义。一般应当在有序序列上使用 `unique()`，不过在无序序列上运行此算法也不是不可以。

基本形式的 `unique()` 会就地运行，不过这个算法还有另一个版本，名为 `unique_copy()`，它会将结果复制到一个新的目标区间。

第 21 章提供了 `list unique()` 算法的一个例子，所以在此不再给出一般形式的例子了。

### 逆置

`reverse()` 算法只是对一个区间中的元素逆序排列。区间中第一个元素会与最后一个元素交换，第二个元素与倒数第二个元素交换，以此类推。

基本形式的 `reverse()` 会就地运行，不过这个算法还有另一个版本，名为 `reverse_copy()`，它会将结果复制到一个新的目标区间。

### 其他修改算法

在本书网站上标准库参考资源中还介绍了另外一些修改算法，包括 `iter_swap()`、`swap_ranges()`、`fill()`、`generate()`、`rotate()`、`next_permutation()` 和 `prev_permutation()`。我们认为这些算法与前面介绍的算法相比，对于每天的工作来说用处没有那么大。不过，如果确实需要使用这些算法，本书网站上标准库参考资源提供了所有详细内容。

## 22.3.4 排序算法

STL 提供了多种版本的排序算法。这些算法并不适用于关联容器，因此关联容器都会在内部对元素排序。另外，`list` 容器提供了自己的 `sort()` 版本，这比通用算法效率更高。因此，大多数排序算法只对 `vector` 和 `deque` 有用。

### 基本排序和合并

`sort()` 函数使用一个快速排序算法对一个区间内的元素排序，一般情况下的时间复杂度是  $O(N \log N)$ 。对一个区间应用 `sort()` 之后，根据 `operator<`，区间内的元素按非递减顺序排列（从最小到最大）。如果不喜欢这个顺序，可以指定一个不同的比较回调，如 `greater`。

`sort()` 有一个变种，名为 `stable_sort()`，它会保持区间中相等元素的相对位置。`stable_sort()` 使用了一个归并排序类的算法。

一旦对区间内的元素进行了排序，就可以应用 `binary_search()` 算法来查找元素，其时间复杂度是对数时间而不是线性时间。



`merge()` 函数允许将两个有序区间合并在一起，并且仍保持有序。其结果是一个包含了两个源区间中所有元素的有序区间。`merge()` 会在线性时间内运行。如果没有 `merge()`，通过连接两个区间，并对连接结果应用 `sort()`，也可以得到同样的效果，不过这样做的效率没有那么多高（将是  $O(N \log N)$  而不是线性时间）。

要确保提供了一个足够大的区间来存储合并之后的结果。

以下是排序和合并的一个例子。

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

// The populateContainer() and print() functions are identical to those
// in the example above, so they are omitted here.

int main(int argc, char** argv)
{
    vector<int> vectorOne, vectorTwo, vectorMerged;
    cout << "Enter values for first vector:\n";
    populateContainer(vectorOne);
    cout << "Enter values for second vector:\n";
    populateContainer(vectorTwo);

    sort(vectorOne.begin(), vectorOne.end());
    sort(vectorTwo.begin(), vectorTwo.end());
    // Make sure the vector is large enough to hold the values
    // from both source vectors.
    vectorMerged.resize(vectorOne.size() + vectorTwo.size());
    merge(vectorOne.begin(), vectorOne.end(), vectorTwo.begin(),
          vectorTwo.end(), vectorMerged.begin());

    cout << "Merged vector: ";
    for_each(vectorMerged.begin(), vectorMerged.end(), &print);
    cout << endl;

    while (true) {
        int num;
        cout << "Enter a number to find (0 to quit): ";
        cin >> num;
        if (num == 0) {
            break;
        }
        if (binary_search(vectorMerged.begin(), vectorMerged.end(), num)) {
            cout << "That number is in the vector.\n";
        } else {
            cout << "That number is not in the vector.\n";
        }
    }

    return (0);
}
```

### 堆排序

堆结构会以一种半有序顺序存储元素，在此结构上，找到最大元素是一个常量时间操作。删除最大元素以及增加一个新元素都为对数时间。有关堆数据结构的一般信息，请参见附录 B 中所列的数据结构参考书。

STL 提供了 4 个算法来处理堆结构：

- `make_heap()` 在线性时间内将一个元素区间转换为一个堆。最大元素即为区间中的第一个元素。
- `push_heap()` 将一个新元素增加到堆中，即把新元素加入到区间原先的末尾位置上（译者注：即最后一个元素的后面）。也就是说，`push_heap()` 要取一个迭代器区间 `[first, last)`，而且希望 `[first, last-1)` 是一个合法的堆，`last-1` 位置上的元素就是要增加到堆中的新元素。对于容器来说，如果在一个 `deque` 容器中有一个堆，可以使用 `push_back()` 向 `deque` 增加新元素，然后在 `deque` 开始和末尾迭代器上调用 `push_heap()`。`push_heap()` 在对数时间内运行。
- `pop_heap()` 从堆删除最大元素，并对余下的元素重新排序以保证仍是一个合法的堆结构。这会使表示堆的区间大小减少一个元素。如果调用前的区间为 `[first, last)`，则新区间为 `[first, last-1)`。与平常一样，这个算法并不会从容器真正删除元素。如果确实要删除容器中的元素，必须在调用 `pop_heap()` 之后调用 `erase()` 或 `pop_back()`。`pop_heap()` 运行时间为对数时间。
- `sort_heap()` 将一个堆区间转换为一个完全有序的区间，时间复杂性为  $O(N \log N)$ 。

堆对于实现优先队列非常有用。实际上，第 21 章介绍的 `priority_queue` 容器就是利用这些堆算法实现的。如果想直接使用堆算法，应当首先看看 `priority_queue` 接口是不是已经能满足要求，只有在不满足的情况下才有必要直接使用堆算法。这里没有提供堆函数的一个例子，不过如果确实需要使用这些算法，本书网站上标准库参考资源包含了有关的详细内容。

### 其他排序例程

还有另外一些排序例程，包括 `partition()`、`partial_sort()` 和 `nth_element()`。它们通常用作快速排序类算法的构建模块。既然 `sort()` 已经提供了一个快速排序类算法，通常并不需要使用这些排序例程。不过，如果确实需要，可以参考本书网站上标准库参考资源提供的详细内容。

### `random_shuffle()`

最后一个“排序”算法从理论上讲更应算是一个“反排序”算法。`random_shuffle()` 会以一个随机顺序重新排列区间中的元素。对于实现诸如洗牌扑克牌等任务，这会很有用。

## 22.3.5 集合算法

STL 中最后一类算法是完成集合操作的 5 个函数。尽管这些算法可以在任何有序迭代器区间上工作，但显然它们所针对的是集合容器的区间。

`includes()` 函数实现了标准子集确定功能，它会检查一个有序区间中的所有元素是否都包括在另一个有序区间中，这些元素可以采用任何顺序出现。

`set_union()`、`set_intersection()`、`set_difference()` 和 `set_symmetric_difference()` 函数实现了这些操作的标准语义。如果你没有研究过集合论，下面简单提一下。两个集合的并集结果是这两个集合中的所有元素。两个集合的交集结果是这两个集合中都有的元素。两个集合的差集结果是在第一个集合中但不在第二个集合中的所有元素。两个集合对称差的结果是集合的“异或”：某一个集合中有而非两个集合中都有的所有元素。



与往常一样，要确保结果区间足够大，能够容纳操作的结果。对于 `set_union()` 和 `set_symmetric_difference()`，结果至多是两个输入区间的大小之和。对于 `set_intersection()` 和 `set_difference()`，结果最多是两个输入区间大小中较大的一个。要记住，不能使用关联容器的迭代器区间（包括集合）来存储结果。

以下是这些算法的一个例子。

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

// The populateContainer() and print() functions are identical to those
// in the example above, so are omitted here.

int main(int argc, char** argv)
{
    vector<int> setOne, setTwo, setThree;
    cout << "Enter set one:\n";
    populateContainer(setOne);
    cout << "Enter set two:\n";
    populateContainer(setTwo);

    // set algorithms work on sorted ranges
    sort(setOne.begin(), setOne.end());
    sort(setTwo.begin(), setTwo.end());

    if (includes(setOne.begin(), setOne.end(), setTwo.begin(), setTwo.end())) {
        cout << "The second set is a subset of the first\n";
    }
    if (includes(setTwo.begin(), setTwo.end(), setOne.begin(), setOne.end())) {
        cout << "The first set is a subset of the second\n";
    }

    setThree.resize(setOne.size() + setTwo.size());
    vector<int>::iterator newEnd;
    newEnd = set_union(setOne.begin(), setOne.end(), setTwo.begin(),
                      setTwo.end(), setThree.begin());
    cout << "The union is: ";
    for_each(setThree.begin(), newEnd, &print);
    cout << endl;

    newEnd = set_intersection(setOne.begin(), setOne.end(), setTwo.begin(),
                              setTwo.end(), setThree.begin());
    cout << "The intersection is: ";
    for_each(setThree.begin(), newEnd, &print);
    cout << endl;

    newEnd = set_difference(setOne.begin(), setOne.end(), setTwo.begin(),
                            setTwo.end(), setThree.begin());
    cout << "The difference between set one and set two is: ";
    for_each(setThree.begin(), newEnd, &print);
    cout << endl;

    newEnd = set_symmetric_difference(setOne.begin(), setOne.end(), setTwo.begin(),
```

```
setTwo.end(), setThree.begin());  
cout << "The symmetric difference is: ";  
for_each(setThree.begin(), newEnd, &print);  
cout << endl;
```

```
return (0);  
}
```

此程序的运行情况如下：

```
Enter set one:  
Enter a number (0 to quit): 5  
Enter a number (0 to quit): 6  
Enter a number (0 to quit): 7  
Enter a number (0 to quit): 8  
Enter a number (0 to quit): 0  
Enter set two:  
Enter a number (0 to quit): 8  
Enter a number (0 to quit): 9  
Enter a number (0 to quit): 10  
Enter a number (0 to quit): 0  
The union is: 5 6 7 8 9 10  
The intersection is: 8  
The difference between set one and set two is: 5 6 7  
The symmetric difference is: 5 6 7 9 10
```

## 22.4 算法和函数对象示例：选民注册审计

选民欺骗可能是美国社会存在的一个问题。有时可能有人会在两个或多个不同的选区注册并投票。另外，重案犯在有些州是禁止投票的，但是可能会以某种方式躲过限制成功地注册和投票。使用新学的算法和函数对象技巧，就可以编写一个简单的选区注册审计函数，它能检查选民注册是否存在反常情况。

### 22.4.1 选民注册审计问题描述

选民注册审计函数要对一个州的信息进行审计。假设选民注册会按区存储在一个 map 中，在此会把区名映射至一个选民 list。审计函数应当取这个 map 和一个重案犯 list 为参数，而且应当从选民 list 中删除所有重案犯。另外，此函数应当找出所有在多个区中注册的选民，而且要把这些选民的名字从所有区的选民列表中删除。为简单起见，在此假设选民 list 只是一个 string 名的 list。实际应用显然还需要更多数据，如地址和党派等。

### 22.4.2 auditVoterRolls() 函数

这个例子将采用一个至顶向下方法，先从最高层函数开始，它会调用尚未编写的函数和函数对象。随着例子的逐步进展，会陆续填入缺少的实现。

顶层函数 auditVoterRolls() 的工作分为三步：

1. 调用 getDuplicates() 来找到所有注册 list 中的所有重复名。

2. 将重复名 list 与重案犯 list 合并，并从合并的 list 中删除重复的元素（译者注：注意这里所删除的是合并后 list 中重复出现的元素，即如果一个人多次注册，而他本身又是一个重案犯，这样在将重复名 list 与重案犯 list 中都有的这个人，反映在最后的合并 list 中就是一个重复元素，这与重复名 list 中的元素即多次注册的人是不同的）。

3. 从每个选民 list 中删除前面所得到合并 list（即由重复名 list 与重案犯 list 合并得到的结果）中的所有名字。这里采用的方法是使用 `for_each()` 来处理 `map` 中的每个 list，并应用一个用户自定义函数对象 `RemoveNames` 从各个 list 中删除有问题的选民名字。

以下是 `auditVoterRolls()` 的实现。

```
//
// auditVoterRolls
//
// Expects a map of string/list<string> pairs keyed on county names
// and containing lists of all the registered voters in those counties
//
// Removes from each list any name on the convictedFelons list and
// any name that is found on any other list
//
void auditVoterRolls(map<string, list<string> >& votersByCounty,
    const list<string>& convictedFelons)
{
    // Get all the duplicate names.
    list<string> duplicates = getDuplicates(votersByCounty);

    // Combine the duplicates and convicted felons--we want
    // to remove names on both lists from all voter rolls.
    duplicates.insert(duplicates.end(), convictedFelons.begin(),
        convictedFelons.end());

    // If there were any duplicates, remove them.
    // Use the list versions of sort and unique instead of the generic
    // algorithms, because the list versions are more efficient.
    duplicates.sort();
    duplicates.unique();

    // Now remove all the names we need to remove.
    for_each(votersByCounty.begin(), votersByCounty.end(),
        RemoveNames(duplicates));
}
```

### 22.4.3 getDuplicates() 函数

`getDuplicates()` 函数必须找到同时在多个选民注册列表中出现的名字。为解决这个问题，有多种不同的方法。这个实现只是将各个区的 list 合并为一个大的 list，并对其排序。此时，不同 list 上出现的重复名字就会连续出现在这个大的 list 中。现在 `getDuplicates()` 就可以在这个大的有序 list 上使用 `adjacent_find()` 算法来找到所有连续出现的重复名。以下是这个函数的实现。

```
//
// getDuplicates()
//
// Returns a list of all names that appear in more than one list in
// the map
//
// The implementation generates one large list of all the names from
// all the lists in the map, sorts it, then finds all duplicates
// in the sorted list with adjacent_find().
//
```

```

list<string> getDuplicates(const map<string, list<string> >& voters)
{
    list<string> allNames, duplicates;

    // Collect all the names from all the lists into one big list.
    map<string, list<string> >::const_iterator it;
    for(it = voters.begin(); it != voters.end(); ++it) {
        allNames.insert(allNames.end(), it->second.begin(), it->second.end());
    }

    // Sort the list--use the list version, not the general algorithm,
    // because the list version is faster.
    allNames.sort();

    //
    // Now that it's sorted, all duplicate names will be next to each other.
    // Use adjacent_find() to find instances of two or more identical names
    // next to each other.
    //
    // Loop until adjacent_find returns the end iterator.
    //
    list<string>::iterator lit;
    for (lit = allNames.begin(); lit != allNames.end(); ++lit) {
        lit = adjacent_find(lit, allNames.end());
        if (lit == allNames.end()) {
            break;
        }
        duplicates.push_back(*lit);
    }

    //
    // If someone was on more than two voter lists, he or she will
    // show up more than once in the duplicates list. Sort the list
    // and remove duplicates with unique.
    //
    // Use the list versions because they are faster than the generic versions.
    //
    duplicates.sort();
    duplicates.unique();

    return (duplicates);
}

```

#### 22.4.4 RemoveNames 函数对象

auditVoterRolls() 函数使用以下代码从选民注册 map 中的各个 list 删除有问题的选民名字（重复注册的选民和重案犯）：

```

for_each(votersByCounty.begin(), votersByCounty.end(),
         RemoveNames(duplicates));

```

for\_each() 算法对 map 中的每个 string/list<string> pair 调用 RemoveNames 函数对象。RemoveNames 函数对象类的定义如下所示：

```

//
// RemoveNames

```



```
//
// Functor class that takes a string/list<string> pair and removes
// any strings from the list that are found in a list of names
// (supplied in the constructor)
//
class RemoveNames : public unary_function<pair<const string, list<string> >,
    void>
{
public:
    RemoveNames(const list<string>& names) : mNameNames(names) {}
    void operator()(pair<const string, list<string> >& val);
protected:
    const list<string>& mNameNames;
};
```

注意，RemoveNames 派生了 unary\_function，这个技术在本章前面介绍过。构造函数取选民姓名 list 的一个引用作为参数，它会保存这个列表以便用于函数调用操作符中。应该记得，函数对象回调的参数是 map 的一个元素，即一个 string/list<string> pair。此函数调用操作符的任务是从 string list 中删除所有出现在 mNameNames list 中的名字。这个实现使用了 remove\_if() 算法，并指定了一个特殊的谓词 NameInList。

```
//
// Function-call operator for RemoveNames functor.
//
// Uses remove_if() followed by erase to actually delete the names
// from the list
//
// Names are removed if they are in our list of mNameNames. Use the NameInList
// functor to check if the name is in the list.
//
void RemoveNames::operator()(pair<const string, list<string> >& val)
{
    list<string>::iterator it = remove_if(val.second.begin(), val.second.end(),
        NameInList(mNameNames));
    val.second.erase(it, val.second.end());
}
```

要记住，remove() 系列算法并不会真正删除元素。它们只是把元素移至区间的最后。必须对容器调用 erase() 才能真正删除元素。

#### 22.4.5 NameInList 函数对象

RemoveNames 函数对象调用了 remove\_if()，并指定了一个 NameInList 类的谓词函数对象。如果作为参数给定的 string 位于 list mNameNames 中，NameInList 函数对象则返回 true。其类定义如下所示：

```
//
// NameInList
//
// Functor to check if a string is in a list of strings (supplied
// at construction time).
//
class NameInList : public unary_function<string, bool>
{
public:
```

```

NameInList(const list<string>& names) : mNameNames(names) {}
bool operator() (const string& val);

```

```

protected:
    const list<string>& mNameNames;
};

```

函数调用操作符的实现只是使用了 `find()` 在 `string` 组成的 `mNames` list 中查找 `name` 参数, 如果 `find()` 返回一个有效的迭代器, 则函数调用操作符返回 `true`; 如果 `find()` 返回末尾迭代器, 函数调用操作符则返回 `false`。

```

//
// function-call operator for NameInList functor
//
// Returns true if it can find name in mNameNames, false otherwise.
// Uses find() algorithm.
//
bool NameInList::operator() (const string& name)
{
    return (find(mNameNames.begin(), mNameNames.end(), name) != mNameNames.end());
}

```

#### 22.4.6 测试 `auditVoterRolls()` 函数

选民注册审计功能的完整实现已经全部给出了。下面是一个很小的测试程序。

```

#include <algorithm>
#include <functional>
#include <map>
#include <list>
#include <iostream>
#include <utility>
#include <string>
using namespace std;

void printString(const string& str)
{
    cout << " (" << str << ")";
}

void printCounty(const pair<const string, list<string> >& county)
{
    cout << county.first << " : ";
    for_each(county.second.begin(), county.second.end(), &printString);
    cout << endl;
}

int main(int argc, char** argv)
{
    map<string, list<string> > voters;
    list<string> nameList, felons;
    nameList.push_back("Amy Aardvark");
    nameList.push_back("Bob Buffalo");
    nameList.push_back("Charles Cat");
    nameList.push_back("Dwayne Dog");
}

```



```
voters.insert(make_pair("Orange", nameList));

nameList.clear();
nameList.push_back("Elizabeth Elephant");
nameList.push_back("Fred Flamingo");
nameList.push_back("Amy Aardvark");

voters.insert(make_pair("Los Angeles", nameList));

nameList.clear();
nameList.push_back("George Goose");
nameList.push_back("Heidi Hen");
nameList.push_back("Fred Flamingo");

voters.insert(make_pair("San Diego", nameList));

felons.push_back("Bob Buffalo");
felons.push_back("Charles Cat");

for_each(voters.begin(), voters.end(), &printCounty);
cout << endl;
auditVoterRolls(voters, felons);
for_each(voters.begin(), voters.end(), &printCounty);

return (0);
}
```

程序的输出如下：

```
Los Angeles: {Elizabeth Elephant} {Fred Flamingo} {Amy Aardvark}
Orange: {Amy Aardvark} {Bob Buffalo} {Charles Cat} {Dwayne Dog}
San Diego: {George Goose} {Heidi Hen} {Fred Flamingo}

Los Angeles: {Elizabeth Elephant}
Orange: {Dwayne Dog}
San Diego: {George Goose} {Heidi Hen}
```

## 22.5 小结

本章将基本 STL 功能介绍完了。在此对可用的各种算法和函数对象提供了一个概述，并介绍了如何编写自己的函数对象。希望你对 STL 容器、算法和函数对象的作用已经有了一定的认识。如果还没有足够的了解，可以想象一下不利用 STL 重新编写上述选民注册审计例子会是什么样子。你要编写自己的 linked-list 和 map 类，还要编写自己的搜索、删除、查找、迭代和其他算法。程序会变得庞大得多，而且较难调试，也较难维护。

如果你对算法和函数对象不以为然，或者认为它们太过复杂，那就没有必要使用这些算法和函数对象。你有充分的自由可以任意选择，如果 find() 算法很适合你的程序，那么不要因为没有用到其他算法也将 find() 算法拒之千里。另外，不要认为 STL 要么全有要么全无。如果只想使用 vector 容器而不想用其他东西，也完全可以只使用 vector 容器。

第 23 章将继续讨论 STL 主题，其中将介绍一些高级特性，包括分配器、迭代器适配器，以及如何编写自己的算法、容器和迭代器。

## 第 23 章 定制和扩展 STL

前两章已经展示了 STL 是一个功能强大的通用容器和算法集。目前为止介绍的内容对于大多数应用来说已经足够了。不过，STL 比前两章所展示的还要灵活得多，而且还可进一步扩展。例如，可以对输入和输出流应用迭代器。可以编写自己的容器、算法和迭代器，甚至可以指定容器使用自己的内存分配机制。本章就会谈到这些高级特性，讨论主要通过介绍如何开发一个新的 STL 容器 `hashmap` 展开的。本章的具体内容包括：

- 更深入地分析分配器
- 迭代器适配器
- 扩展 STL
  - 编写算法
  - 编写容器：一个散列映射实现
  - 编写迭代器：一个散列映射迭代器实现

如果胆量不够，就不适合阅读本章！本章会深入探讨 C++ 语言中最复杂而且语法上最容易混淆的方面。如果觉得前两章介绍的基本 STL 容器和算法已经够用了，可以跳过这一章。不过，如果确实想了解 STL，而不只是用用它而已，那就应该阅读这一章。在看本章之前，一定要充分掌握第 11 章介绍的模板内容。

### 23.1 分配器

第 21 章提到，每个 STL 容器都取一个 `Allocator` 类型作为一个模板参数，其默认值往往就足够了。例如，`vector` 模板定义如下所示。

```
template <typename T, typename Allocator = allocator<T> > class vector;
```

容器构造函数允许指定一个类型为 `Allocator` 的对象。利用这些额外的参数，可以对容器分配内存的方式进行定制。容器完成的每个内存分配都是通过调用 `Allocator` 对象的 `allocate()` 方法完成的。与之相对，每个撤销都是通过调用 `Allocator` 对象的 `deallocate()` 方法完成的。标准库提供了一个名为 `allocator` 的默认 `Allocator` 类，它只是将这些方法（`allocate()` 和 `deallocate()`）实现为 `operator new` 和 `operator delete` 的包装器。

如果希望程序中的容器使用一个定制的内存分配和撤销机制，如内存池，就可以编写自己的 `Allocator` 类。只要一个类提供了 `allocate()`、`deallocate()` 和其他一些必要的方法和 `typedef`，都可以用于取代默认的 `allocator` 类。不过，从我们的经验看来，很少会用到这个特性，所以本书不介绍有关的细节。要了解更多详细内容，请参考附录 B 中所列的有关 C++ 标准库的参考书。

## 23.2 迭代器适配器

标准库提供了三个迭代器适配器 (iterator adapter)，这是建立在其他迭代器之上的特殊迭代器。在第 26 章你将了解到有关适配器设计模式的更多内容。对现在来说，只要知道这些迭代器能做什么就可以了。上述三个迭代器适配器都在 `<iterator>` 头文件中声明。

还可以编写自己的迭代器适配器。详细内容请参考附录 B 所列的有关标准库的参考书。

### 23.2.1 逆序迭代器

STL 提供了一个 `reverse_iterator` 类，它会通过一个双向或随机访问迭代器逆向地迭代处理容器中的元素。对 `reverse_iterator` 应用 `operator++` 会对底层容器迭代器调用 `operator--`，反之亦然，即对 `reverse_iterator` 应用 `operator--` 会对底层容器迭代器调用 `operator++`。STL 中的每个可逆容器 (reversible container) 恰好是属于标准部分的所有容器，它们都提供了一个 `typedef reverse_iterator` 以及 `rbegin()` 和 `rend()` 等方法。`rbegin()` 返回从容器最后一个元素开始的一个 `reverse_iterator`，`rend()` 返回从容器第一个元素开始的一个 `reverse_iterator`。

STL 中有些算法没有相应的以逆序工作的算法，`reverse_iterator` 主要是对这些算法最为有用。例如，基本 `find()` 算法会搜索序列中的第一个元素。如果想找到序列中的最后一个元素，就可以使用 `reverse_iterator`。需要注意，在对 `reverse_iterator` 调用一个诸如 `find()` 的算法时，它也会返回一个 `reverse_iterator`。通过对 `reverse_iterator` 调用 `base()` 方法，总能从一个 `reverse_iterator` 得到一个正常的 `iterator`。不过，按照 `reverse_iterator` 的实现细节，从 `base()` 返回的 `iterator` 所指示的元素并不是调用此方法的 `reverse_iterator` 所指示的元素，而是 `reverse_iterator` 所指示元素后面的那个元素。

以下是对一个 `reverse_iterator` 调用 `find()` 的例子。

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>
using namespace std;

// The implementation of populateContainer() is identical to that shown in
// Chapter 22, so it is omitted here.

int main(int argc, char** argv)
{
    vector<int> myVector;
    populateContainer(myVector);

    int num;
    cout << "Enter a number to find: ";
    cin >> num;

    vector<int>::iterator it1;
    vector<int>::reverse_iterator it2;
    it1 = find(myVector.begin(), myVector.end(), num);
    it2 = find(myVector.rbegin(), myVector.rend(), num);

    if (it1 != myVector.end()) {
```

```

        cout << "Found " << num << " at position " << it1 - myVector.begin()
            << " going forward.\n";
        cout << "Found " << num << " at position "
            << it2.base() - 1 - myVector.begin() << " going backward.\n";
    } else {
        cout << "Failed to find " << num << endl;
    }

    return (0);
}

```

这个程序中有一行代码需要多做一些解释。为了打印逆序迭代器找到的位置，相应代码如下：

```

        cout << "Found " << num << " at position "
            << it2.base() - 1 - myVector.begin() << " going backward.\n";

```

如前所述，base() 返回一个 iterator，它指示了 reverse\_iterator 所指示元素后面的一个元素。为了指示同一个元素，必须将 base() 返回的 iterator 减 1。

### 23.2.2 流迭代器

第 21 章曾经提到，STL 提供了一些迭代器，可以把输入和输出流看作是输入和输出迭代器。使用这些迭代器，就可以对输入和输出流进行适配，这样输入和输出流可以在各种 STL 算法中分别用作源和目标。例如，可以使用 ostream\_iterator 利用 copy() 算法打印出一个容器中的元素，这只需一行代码：

```

#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int main(int argc, char** argv)
{
    vector<int> myVector;
    for (int i = 0; i < 10; i++) {
        myVector.push_back(i);
    }

    // Print the contents of the vector.
    copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}

```

ostream\_iterator 是一个模板类，它取元素类型作为一个类型参数。其构造函数取一个输出流和一个 string 作为参数，该 string 是在各元素之后写至流的字符串。（译者注：由于这里 string 参数为“ ”，其作用是在输出的各元素之间用空格分隔开）。

类似地，可以使用迭代器抽象利用 istream\_iterator 从一个输入流读取值。istream\_iterator 可以在算法和容器方法中用作源。它不如 ostream\_iterator 那么常用，所以这里不再提供输入流迭代器的例子。有关详细内容请参考附录 B 所列的参考资料。

### 23.2.3 插入迭代器

第 22 章曾经介绍过，诸如 copy() 等算法不会把元素插入到一个容器中，它们只是把一个区间中的老元素代之以新元素。为了让 copy() 之类的算法更有用，STL 提供了三个插入迭代器，用以真正将元素插



人到容器中。插入迭代器基于容器类型模板化，并在构造函数中取实际的容器引用作为参数。通过提供必要的迭代器接口，这些适配器可以用作 `copy()` 等算法的目标迭代器。不过，这些适配器并不是替换容器中的元素，而是会对容器做调用，从而真正插入新元素。

基本 `insert_iterator` 对容器调用 `insert(position, element)`，`back_insert_iterator` 对容器调用 `push_back(element)`，`front_insert_iterator` 则对容器调用 `push_front(element)`。

例如，可以使用 `back_insert_iterator` 利用 `remove_copy_if()` 算法在一个新 `vector` 中填入一个老 `vector` 中所有不等于 100 的元素：

```
#include <algorithm>
#include <functional>
#include <iterator>
#include <vector>
#include <iostream>

using namespace std;

// The implementation of populateContainer() is identical to that shown in
// Chapter 22, so it is omitted here.
int main(int argc, char** argv)
{
    vector<int> vectorOne, vectorTwo;
    populateContainer(vectorOne);

    back_insert_iterator<vector<int> > inserter(vectorTwo);
    remove_copy_if(vectorOne.begin(), vectorOne.end(), inserter,
        bind2nd(equal_to<int>(), 100));

    copy(vectorTwo.begin(), vectorTwo.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return (0);
}
```

可以看到，在使用插入迭代器时，不需要提前确定目标容器的大小。

`insert_iterator` 和 `front_insert_iterator` 的功能类似，不过 `insert_iterator` 在其构造函数中还取一个初始的迭代器位置，这个位置会传递给第一个 `insert(position, element)` 调用。以后会基于各 `insert()` 调用的返回值生成后续的迭代器位置提示。

`insert_iterator` 有一个很大的好处，利用它就可以使用关联容器作为修改算法的目标。还记得第 22 章曾指出，关联容器的问题在于，不允许对迭代处理的元素进行修改。通过使用一个 `insert_iterator`，可以插入元素，并允许容器在内部对其适当地排序。第 21 章介绍过，关联容器实际上支持一种取迭代器位置作为参数的 `insert()`，它会把这个位置用作一个“提示”，因此有可能并不采用所提示的这个位置，只是将其忽略。在对一个关联容器使用 `insert_iterator` 时，可以传递容器的开始或末尾迭代器用作提示。以下对前面的例子做了修改，在此目标容器是一个 `set` 而不是一个 `vector`：

```
#include <algorithm>
#include <functional>
#include <iterator>
#include <vector>
#include <iostream>
#include <set>
```

```
using namespace std;

// The implementation of populateContainer() is identical to that shown in
// Chapter 22, so it is omitted here.

int main(int argc, char** argv)
{
    vector<int> vectorOne;
    set<int> setOne;

    populateContainer(vectorOne);

    insert_iterator<set<int>> inserter(setOne, setOne.begin());

    remove_copy_if(vectorOne.begin(), vectorOne.end(), inserter,
        bind2nd(equal_to<int>(), 100));

    copy(setOne.begin(), setOne.end(), ostream_iterator<int>(cout, " "));

    cout << endl;

    return (0);
}
```

需要注意，`insert_iterator` 每次调用 `insert()` 后会修改传至 `insert()` 的迭代器位置提示，使这个位置总是刚才所插入元素的后面一个位置。

## 23.3 扩展 STL

STL 包括许多有用的容器、算法和迭代器，可以在应用中自由使用。不过，一个库要想包括所有可能的工具来满足所有潜在客户的需求，这是不可能的。因此，最好的库应当是可扩展的，这些库允许客户对基本功能进行调整和增补，以得到自己所需要的那个功能。STL 本质上是可扩展的，因为其基本结构就是将数据与操作数据的算法相分离。可以编写自己的容器，只需提供一个遵循 STL 标准的迭代器，你的容器就可以用于 STL 算法。类似地，也可以编写一个函数用于标准容器的迭代器。这一节将解释扩展 STL 的基本原则，并提供一些扩展的示例实现。

### 23.3.1 为什么要扩展 STL

如果你要用 C++ 编写一个算法或容器，可以遵循 STL 约定，也可以不遵循。对于简单的容器和算法，花功夫遵循 STL 原则可能有些得不偿失。不过，对于你打算重用的重要代码，为此投入的努力是有意义的。首先，其他 C++ 程序员可以更容易地读懂这样的代码，因为你遵循了定义明确的接口原则。其次，你能在 STL 的其他部分之上（算法或容器）利用你的容器或算法，而无需提供特殊的修补或适配器。最后一点，这样做会要求你本着严格的方针开发代码，这是开发可靠代码所必需的。

### 23.3.2 编写 STL 算法

第 22 章介绍的算法很有用，不过不可避免地，你可能会在程序中遇到需要一些新算法的情况。倘若如此，可以编写自己的算法，让这些算法像标准算法一样用于 STL 迭代器，这并不难。

`find_all()`

假设你想找出一个给定区间中与一个谓词匹配的所有元素。`find()` 和 `find_if()` 是最可选的算法，但是这两个算法都只是返回仅指示一个元素的迭代器。实际上，没有任何标准算法能够找出与一个谓词匹配的所有元素。好在可以针对此功能编写自己的版本，名为 `find_all()`。

第一个任务是定义函数原型。可以仿照 `find_if()` 来建立 `find_all()` 的原型。这是一个基于两类参数（迭代器和谓词）的模板化函数。这两类参数是开始和末尾迭代器以及谓词对象。它与 `find_if()` 的区别只



是返回值有所不同：find\_if()会返回指示匹配元素的一个迭代器，find\_all()则返回指示所有匹配元素的一个迭代器向量。以下是函数原型：

```
template <typename InputIterator, typename Predicate>
vector<InputIterator>
find_all(InputIterator first, InputIterator last, Predicate pred);
```

还有一个选择，可以返回一个迭代器来迭代处理容器中的所有匹配元素，不过这需要编写自己的迭代器类。

下一个任务是编写实现。find\_all()可以建立在 find\_if()之上，即重复地调用 find\_if()。第一次调用 find\_if()时使用所提供的从 first 到 last 的完整区间。第二个调用使用一个小一点的区间，即从上一次调用所找到的元素开始到 last。如此继续循环，直到 find\_if()无法找到匹配元素为止。这个函数实现如下：

```
{
    vector<InputIterator> res;

    while (true) {
        // Find the next match in the current range.
        first = find_if(first, last, pred);
        // check if find_if failed to find a match
        if (first == last) {
            break;
        }
        // Store this match.
        res.push_back(first);
        // Shorten the range to start at one past the current match
        ++first;
    }
    return (res);
}
```

下面的代码用以测试这个函数：

```
int main(int argc, char** argv)
{
    int arr[] = {3, 4, 5, 4, 5, 6, 5, 8};
    vector<int*> all = find_all(arr, arr + 8, bind2nd(equal_to<int>(), 5));

    cout << "Found " << all.size() << " matching elements: ";

    for (vector<int*>::iterator it = all.begin(); it != all.end(); ++it) {
        cout << **it << " ";
    }
    cout << endl;

    return (0);
}
```

上述测试代码有点难理解，所以下面做一些解释。这个测试使用一个 int 数组作为一个 STL 容器。在第 21 章曾说过，C 风格的数组也是合法的容器，可以把指针作为迭代器。数组的开始迭代器就是指向第一个元素的指针。末尾迭代器是指向最后一个元素后面一个位置的指针。

找到所有元素的迭代器后，测试代码会统计所找到的元素个数，这就是 all vector 中的迭代器个数。然后，它迭代处理 all vector，打印出每个元素。需要注意 it 的双重解除引用：第一次解除引用得到 int\*，第二次解除引用得到实际的 int。

### 迭代器特性

有些算法实现需要有关迭代器的额外信息。例如，可能需要知道迭代器所指示元素的类型，以便存储临时值，或者算法可能想知道迭代器是双向迭代器还是随机访问迭代器。

C++ 提供了一个名为 `iterator_traits` 的类模板，由此可以找到迭代器的信息。可以用你感兴趣的迭代器类型对 `iterator_traits` 类模板实例化，并访问以下 `typedef` 之一：`value_type`、`difference_type`、`iterator_category`、`pointer` 和 `reference`。例如，以下模板函数声明了一个临时变量，其类型即为 `IteratorType` 类型迭代器所指示的类型：

```
#include <iterator>

template <typename IteratorType>
void iteratorTraitsTest(IteratorType it)
{
    typename std::iterator_traits<IteratorType>::value_type temp;
    temp = *it;
    cout << temp << endl;
}
```

要注意，这里在 `iterator_traits` 代码行前面使用了关键字 `typename`。第 21 章曾经做过解释，如果要基于一个或多个模板参数来访问一个类型，就必须显式地指定关键字 `typename`。在这个例子中，要用模板参数 `IteratorType` 来访问 `value_type` 类型。

### 23.3.3 编写一个 STL 容器

C++ 标准对容器提出了许多需求，只有满足这些需求才能算得上是 STL 容器。另外，如果你希望容器是顺序容器（像一个 `vector`）或关联容器（像一个 `map`），还必须遵循补充需求。

在编写一个新的容器时，我们的建议是，先编写遵循一般 STL 规则的基本容器，如让它作为一个类模板，但是不要太操心是否遵循 STL 的特定细节。开发完实现后，可以再增加迭代器和方法，使之能够用于 STL 框架。这一节就采用这种方法来开发一个散列映射（`hashmap`）。

#### 基本 `hashmap`

STL 中最大的疏忽就是少了一个散列表容器。STL `map` 和 `set` 能提供对数时间的插入、查找和删除，与之不同，散列表在平均情况下可以提供常量时间的插入、删除和查找。散列表并不是将元素有序地存储，而是把各元素散列（`hash`）或映射至某个桶（`bucket`）。只要所存储的元素个数不比桶数大太多，插入、删除和查找操作都能在常量时间内运行。

本节假设你对散列数据结构已经很熟悉。如果不了解这个内容，可以参考附录 B 中列出的有关标准数据结构的资料。

许多 STL 的特定实现提供了一个非标准的散列表。不过，可以想到，由于缺乏标准化，每个实现都会稍有不同。这一节将提供一个简单但功能完备的 `hashmap` 实现，这个 `hashmap` 可以在多个平台上使用。类似于 `map`，`hashmap` 也存储键/值对。实际上，它提供的操作与 `map` 提供的操作几乎一样。

这个 `hashmap` 实现使用链式散列（也称开放散列），而且没有提供诸如再散列等高级特性。

#### 散列函数

编写一个 `hashmap` 时，第一个要做的选择是如何处理散列函数。有一句话说得好，好的抽象会使容易的情况更容易，使难的情况也有可能解决。好的 `hashmap` 接口允许客户指定自己的散列函数和桶数，从而针对其特定工作负载定制散列行为。另一方面，客户如果没有这种需求，或者没有能力来编写一个

好的散列函数以及选择桶数，也应该能够不做任何定制地直接使用容器。一种解决方案是允许客户在 `hashmap` 构造函数中提供散列函数和桶数，并为之提供默认值。另外，把散列函数和桶数包装在一个散列类中也是一种合理的做法。我们的默认散列类定义如下所示：

```
// Any Hash Class must provide two methods: hash() and numBuckets().
template <typename T>
class DefaultHash
{
public:
    // Throws invalid_argument if numBuckets is nonpositive
    DefaultHash(int numBuckets = 101) throw (invalid_argument);
    int hash(const T& key) const;
    int numBuckets() const { return mNumBuckets; }

protected:
    int mNumBuckets;
};
```

注意 `DefaultHash` 类针对其散列的键类型模板化，以便支持一个模板化的 `hashmap` 容器。构造函数的实现很简单：

```
// Throws invalid_argument if numBuckets is nonpositive
template <typename T>
DefaultHash<T>::DefaultHash(int numBuckets) throw (invalid_argument)
{
    if (numBuckets <= 0) {
        throw (invalid_argument("numBuckets must be > 0"));
    }
    mNumBuckets = numBuckets;
}
```

`hash()` 的实现要难一些，部分原因在于它必须应用于任何类型的键。它要把键映射至某个 `mNumBuckets` 桶。此函数使用了除留余教法 (division method) 来完成散列，即键所对应的桶 (桶号) 是键对桶数取余数后得到的一个整数值。

```
// Uses the division method for hashing.
// Treats the key as a sequence of bytes, sums the ASCII
// values of the bytes, and mods the total by the number
// of buckets.
template <typename T>
int DefaultHash<T>::hash(const T& key) const
{
    int bytes = sizeof(key);
    unsigned long res = 0;
    for (int i = 0; i < bytes; ++i) {
        res += *((char*)&key + i);
    }
    return (res % mNumBuckets);
}
```

遗憾的是，前面的方法无法应用于 `string`，因为不同的 `string` 对象可能包含相同的 `string` 值。因此，相同的 `string` 值就可能散列到不同的桶中。这样一来，可以想到，专门针对 `string` 提供 `DefaultHash` 类的一个部分特殊化应该是个不错的想法：

```

// Specialization for strings
template <>
class DefaultHash<string>
{
public:
    // Throws invalid_argument if numBuckets is nonpositive
    DefaultHash(int numBuckets = 101) throw (invalid_argument);
    int hash(const string& key) const;
    int numBuckets() const { return mNumBuckets; }

protected:
    int mNumBuckets;
};

// Throws invalid_argument if numBuckets is nonpositive
DefaultHash<string>::DefaultHash(int numBuckets) throw (invalid_argument)
{
    if (numBuckets <= 0) {
        throw (invalid_argument("numBuckets must be > 0"));
    }
    mNumBuckets = numBuckets;
}

// Uses the division method for hashing after summing the
// ASCII values of all the characters in key.
int DefaultHash<string>::hash(const string& key) const
{
    int sum = 0;

    for (size_t i = 0; i < key.size(); i++) {
        sum += key[i];
    }
    return (sum % mNumBuckets);
}

```

如果客户想使用其他指针类型或对象作为键，他就应该为这些类编写自己的散列类。

本节所示的散列函数只是针对基本 hashmap 实现的简单例子。它们不能保证在全键范围做到均匀的散列。如果需要数学上更严格的散列函数（或者不知道什么是“均匀散列”），可以参考有关算法的资料。

#### hashmap 接口

hashmap 支持 3 个基本操作：插入、删除和查找。当然，它也提供了构造函数、析构函数、复制构造函数和赋值操作符。以下是 hashmap 类模板的公共部分：

```

template <typename Key, typename T, typename Compare = std::equal_to<Key>,
          typename Hash = DefaultHash<Key> >
class hashmap
{
public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;

    // Constructors
    // Throws invalid_argument if the hash object specifies a nonpositive

```



```

// number of buckets
explicit hashmap(const Compare& comp = Compare(),
    const Hash& hash = Hash()) throw(invalid_argument);

// destructor, copy constructor, assignment operator
~hashmap();
hashmap(const hashmap<Key, T, Compare, Hash>& src);
hashmap<Key, T, Compare, Hash>& operator=(
    const hashmap<Key, T, Compare, Hash>& rhs);

// Element insert
// Inserts the key/value pair x
void insert(const value_type& x);

// Element delete
// Removes the element with key x, if it exists
void erase(const key_type& x);

// Element lookup
// find returns a pointer to the element with key x.
// Returns NULL if no element with that key exists.
value_type* find(const key_type& x);

// operator[] finds the element with key x or inserts an
// element with that key if none exists yet. Returns a reference to the
// value corresponding to that key.
T& operator[] (const key_type& x);

protected:
    // Implementation details not shown yet
};

```

可以看到, key 和 value 类型都是模板参数, 这与 STL map 中类似。hashmap 将 `pair<const Key, T>` 作为实际元素存储在容器中。insert()、erase()、find() 和 operator [] 方法都很简单。不过, 这个接口还有几个方面需要进一步说明。

#### 比较模板参数

类似于 map、set 和其他标准容器, hashmap 允许客户指定比较类型作为模板参数, 并且能在构造函数中传递该类的一个特定比较对象。但与 map 和 set 不同, hashmap 不会按键对元素排序, 而是必须比较键的相等性。因此, 这里不使用 less 作为默认比较操作, 它使用的是 equal\_to。比较对象只是用于检测是否试图向容器中插入重复的键。

#### 散列模板参数

如果允许客户定义自己的类, 以便构造此类对象传入构造函数, 此时必须明确如何在构造函数中指定该参数的类型。为此有多种方法。STL 的方法是最复杂的一种, 它取类的类型作为一个模板参数, 并使用该模板化类型作为构造函数中的类型。可以看到, 对于这个散列类, 我们就采用了这种方法。因此, hashmap 模板取 4 个模板参数: 键类型、值类型、比较类型和散列类型。

#### typedef

hashmap 类模板定义了 3 个 typedef:

```

typedef Key key_type;
typedef T mapped_type;
typedef pair<const Key, T> value_type;

```

特别地, `value_type` 对于引用比较麻烦的 `pair<const Key, T>` 非常有用。

可以看到, 根据标准, 这些 `typedef` 也是 STL 容器所必需的。

#### 实现

完成了 `hashmap` 接口后, 下面该选择实现模型了。基本散列表结构通常包括固定数目的桶, 每个桶可以存储一个或多个元素。应当能够基于一个 `bucket-id` (对键应用散列函数计算得到的结果) 在常量时间内访问桶。因此, `vector` 是对桶最为适合的容器。每个桶必须存储一个元素列表, 因此 STL `list` 可以用作桶类型。这样得到的最后结构是: `pair<const Key, T>` 元素 `list` 的一个 `vector` (译者注: 即散列表结构是一个包括 `list` 的 `vector`, 而 `list` 中包含的元素为 `pair<const Key, T>`)。以下是 `hashmap` 类的 `protected` 成员:

```
protected:
    typedef list<value_type> ListType;

    // In this first implementation, it would be easier to use a vector
    // instead of a pointer to a vector, which requires dynamic allocation.
    // However, we use a ptr to a vector so that, in the final
    // implementation, swap() can be implemented in constant time.
    vector<ListType>* mElems;
    int mSize;
    Compare mComp;
    Hash mHash;
```

如果没有 `value_type` 和 `ListType` 的 `typedef`, 声明 `mElems` 的那一行可能要写作:

```
vector<list<pair<const Key, T> > >* mElems;
```

`mComp` 和 `mHash` 成员分别存储比较和散列对象, `mSize` 存储当前在容器中的元素个数。

#### 构造函数

构造函数要初始化所有字段, 并分配一个新的 `vector`。遗憾的是, 这个模板语法有些复杂。如果你不清楚这个语法, 可以参见第 11 章来了解编写类模板的有关细节。

```
// Construct mElems with the number of buckets.
template <typename Key, typename T, typename Compare, typename Hash>
hashmap<Key, T, Compare, Hash>::hashmap(
    const Compare& comp, const Hash& hash) throw(invalid_argument) :
    mSize(0), mComp(comp), mHash(hash)
{
    if (mHash.numBuckets() <= 0) {
        throw (invalid_argument("Number of buckets must be positive"));
    }
    mElems = new vector<list<value_type> >(mHash.numBuckets());
}
```

这个实现至少需要一个桶, 所以构造函数要求必须满足这个限制。

#### 析构函数、复制构造函数和赋值操作符

只有 `mElems` 数据成员需要撤销、复制和赋值。以下是析构函数、复制构造函数和赋值操作符的实现:

```
template <typename Key, typename T, typename Compare, typename Hash>
hashmap<Key, T, Compare, Hash>::~hashmap()
{
}
```



```

    delete mElems;
}

template <typename Key, typename T, typename Compare, typename Hash>
hashmap<Key, T, Compare, Hash>::hashmap(
    const hashmap<Key, T, Compare, Hash>& src) :
    mSize(src.mSize), mComp(src.mComp), mHash(src.mHash)
{
    // Don't need to bother checking if numBuckets is positive, because
    // we know src checked

    // Use the vector copy constructor.
    mElems = new vector<list<value_type> >(*(src.mElems));
}

template <typename Key, typename T, typename Compare, typename Hash>
hashmap<Key, T, Compare, Hash>& hashmap<Key, T, Compare, Hash>::operator=(
    const hashmap<Key, T, Compare, Hash>& rhs)
{
    // Check for self-assignment.
    if (this != &rhs) {
        delete mElems;
        mSize = rhs.mSize;
        mComp = rhs.mComp;
        mHash = rhs.mHash;
        // Don't need to bother checking if numBuckets is positive, because
        // we know rhs checked

        // Use the vector copy constructor.
        mElems = new vector<list<value_type> >(*(rhs.mElems));
    }
    return (*this);
}

```

要注意，复制构造函数和赋值操作符都会使用 vector 的复制构造函数，以源 hashmap 中的 vector 作为源来构造一个新的 vector。

### 元素查找

三大主要操作（查找、插入和删除）都需要有相应代码来根据一个给定键查找元素。因此，如果一个 protected 辅助方法来完成这个任务会很有帮助。findElement() 首先使用散列对象将键散列至一个特定的桶。然后，在该桶中查找是否有元素的键与给定键匹配。所存储的元素是键/值对，所以具体的比较会在元素第一个字段上进行。构造函数中指定的比较函数对象就是用于完成这个比较。list 需要完成线性查找来寻找匹配的元素，因此可以使用 find() 算法而不是一个显式的 for 循环。

```

template <typename Key, typename T, typename Compare, typename Hash>
typename list<pair<const Key, T> >::iterator
hashmap<Key, T, Compare, Hash>::findElement(const key_type& x, int& bucket) const
{
    // Hash the key to get the bucket.
    bucket = mHash.hash(x);

    // Look for the key in the bucket.
    for (typename ListType::iterator it = (*mElems)[bucket].begin();
         it != (*mElems)[bucket].end(); ++it) {
        if (mComp(it->first, x)) {

```

```

        return (it);
    }
    return ((*mElems)[bucket].end());
}

```

要注意，findElement() 返回一个指示 list 中某元素的迭代器，该元素表示键散列到的那个桶。如果找到了这个元素，迭代器就指示该元素，否则，迭代器则是该 list 的末尾迭代器。这个桶按引用在桶参数中返回。

这个方法中的语法有点让人搞不懂，特别是这里使用了 typename 关键字。在第 21 章曾经做过解释，如果要使用依赖于一个（译者注：或多个）模板参数的类型，就必须使用 typename 关键字。具体地，list<pair<const Key, T>>::iterator 类型就依赖于 Key 和 T 模板参数。

这个语法中还有一点需要注意，mElems 是一个指针，所以在对它应用 operator [] 来得到一个特定的元素之前，必须对其解除引用。因此就有了上述有点难看的 (\*mElems)[bucket]。

可以把 find() 方法实现为 findElement() 的一个简单包装器：

```

template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::value_type*
hashmap<Key, T, Compare, Hash>::find(const key_type& x)
{
    int bucket;
    // Use the findElement() helper.
    typename ListType::iterator it = findElement(x, bucket);
    if (it == (*mElems)[bucket].end()) {
        // We didn't find the element--return NULL.
        return (NULL);
    }
    // We found the element. Return a pointer to it.
    return (&(*it));
}

```

operator [] 方法与之类似，只不过如果无法找到匹配元素，它会将元素插入：

```

template <typename Key, typename T, typename Compare, typename Hash>
T& hashmap<Key, T, Compare, Hash>::operator[] (const key_type& x)
{
    // Try to find the element.
    // If it doesn't exist, add a new element.
    value_type* found = find(x);
    if (found == NULL) {
        insert(make_pair(x, T()));
        found = find(x);
    }
    return (found->second);
}

```

这个方法的效率不太高，因为在最坏情况下，它会调用两次 find() 和一次 insert()。不过，这些操作都在常量时间内运行，而不论 hashmap 中有多少元素。所以开销也不是太大。

### 元素插入

insert() 必须首先检查 hashmap 中是否已经存在此键的元素。如果没有，会把元素插入到适当的桶中的 list。需要说明，findElement() 会按引用返回键所散列到的那个桶，即使在该桶中并没有找到此键的

元素。

```
template <typename Key, typename T, typename Compare, typename Hash>
void hashmap<Key, T, Compare, Hash>::insert(const value_type& x)
{
    int bucket;
    // Try to find the element.
    typename ListType::iterator it = findElement(x.first, bucket);

    if (it != (*mElems)[bucket].end()) {
        // The element already exists.
        return;
    } else {
        // We didn't find the element, so insert a new one.
        mSize++;
        (*mElems)[bucket].insert((*mElems)[bucket].end(), x);
    }
}
```

### 元素删除

erase()与insert()的模式一样，首先调用 findElement()试图找到元素。如果元素存在，则从适当的桶中的列表将元素删除。否则什么也不做。

```
template <typename Key, typename T, typename Compare, typename Hash>
void
hashmap<Key, T, Compare, Hash>::erase(const key_type& x)
{
    int bucket;

    // First, try to find the element.
    typename ListType::iterator it = findElement(x, bucket);

    if (it != (*mElems)[bucket].end()) {
        // The element already exists--erase it.
        (*mElems)[bucket].erase(it);
        mSize--;
    }
}
```

### 使用基本 hashmap

以下是一个小测试程序来展示如何使用这个基本 hashmap 类模板。

```
#include "hashmap.h"

int main(int argc, char** argv)
{
    hashmap<int, int> myHash;
    myHash.insert(make_pair(4, 40));
    myHash.insert(make_pair(6, 60));

    hashmap<int, int>::value_type* x = myHash.find(4);
    if (x != NULL) {
        cout << "4 maps to " << x->second << endl;
    } else {
```

```

        cout << "cannot find 4 in map\n";
    }

    myHash.erase(4);

    hashmap<int, int>::value_type* x2 = myHash.find(4);
    if (x2 != NULL) {
        cout << "4 maps to " << x2->second << endl;
    } else {
        cout << "cannot find 4 in map\n";
    }

    myHash[4] = 35;

    return (0);
}

```

其输出为：

```

4 maps to 40
cannot find 4 in map

```

### 将 hashmap 作为一个 STL 容器

刚才所示的基本 hashmap 与 STL 的宗旨是一致的，但并非逐字逐句都符合。对于大多数用途来说，前面的实现已经够好了。不过，如果在这个散列映射上使用 STL 算法，还要多做一些工作。C++ 标准指定了一些特定的方法和 typedef，如果一个数据结构要作为一个容器，就必须提供这些方法和 typedef。

#### 有关 typedef 的容器需求

表 23-1 列出了需要提供的 typedef。

表 23-1

类 型 名	说 明
value_type	容器中存储的元素类型
reference	容器中存储的元素类型的一个引用
const_reference	容器中存储的 const 元素类型的一个引用
iterator	迭代处理容器中元素的“智能指针”类型
const_iterator	iterator 的另一个版本，用以迭代处理容器中的 const 元素
size_type	可以表示容器中元素个数的类型，通常就是 size_t（在 <cstdlib> 中声明）
difference_type	可以表示容器的两个 iterator 之差 的类型，通常就是 ptrdiff_t（在 <cstdlib> 中声明）

以下是 hashmap 类中所有这些 typedef 的定义（除了 iterator 和 const\_iterator）。在后面一节中将详细介绍如何编写一个迭代器。要注意，value\_type（再加上 key\_type 和 mapped\_type，将在后面讨论）已经在前一个版本的 hashmap 中定义过。

```

template <typename Key, typename T, typename Compare = std::equal_to<Key>,
          typename Hash = DefaultHash<Key> >
class hashmap
{

```

```

public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
    typedef pair<const Key, T>& reference;
    typedef const pair<const Key, T>& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    // Remainder of class definition omitted for brevity
};

```

### 有关方法的容器需求

除了 typedef, 表 23-2 列出了每个容器都必须提供的方法。

表 23-2

方 法	说 明	复 杂 性
默认构造函数	构造一个空的容器	常量时间
复制构造函数	完成一个深复制	线性时间
赋值操作符	完成一个深复制	线性时间
析构函数	撤销动态分配的内存, 对容器中剩余的所有元素调用析构函数	线性时间
iterator begin(); const_iterator begin()const;	返回指示容器中第一个元素的一个迭代器	常量时间
iterator end(); const_iterator end()const;	返回指示容器中最后一个元素的一个迭代器	常量时间
operator== operator!= operator< operator> operator<= operator>=	对两个容器逐元素进行比较的比较操作符	线性时间
void swap (Container&);	将传递至方法的容器的内容与调用此方法的对象进行交换	常量时间 (不过理论上讲, 标准只是说“应该”如此)
size_type size()const;	返回容器中的元素个数	常量时间 (不过理论上讲, 标准只是说“应该”如此)
size_type max_size()const;	返回容器可以容纳的最大元素个数	常量时间 (不过理论上讲, 标准只是说“应该”如此)
bool empty()const;	指定容器是否包含元素	常量时间

在这个 hashmap 例子中, 我们忽略了比较操作符。实现比较操作符是一个很好的练习, 这个练习留给读者来完成!

以下是余下所有方法 (除 begin() 和 end() 之外) 的声明和定义。begin() 和 end() 将在下面的“编写一个迭代器”小节介绍。



```

template <typename Key, typename T, typename Compare = std::equal_to<Key>,
          typename Hash = DefaultHash<Key> >
class hashmap
{
public:
    // typedefs omitted for brevity
    // Constructors
    explicit hashmap(const Compare& comp = Compare(),
                    const Hash& hash = Hash()) throw(invalid_argument);
    // destructor, copy constructor, assignment operator
    ~hashmap();
    hashmap(const hashmap<Key, T, Compare, Hash>& src);
    hashmap<Key, T, Compare, Hash>& operator=(
        const hashmap<Key, T, Compare, Hash>& rhs);

    // Size methods
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    // Other modifying utilities
    void swap(hashmap<Key, T, Compare, Hash>& hashIn);

    // Other methods omitted for brevity
};

```

The implementations of the constructor, destructor, copy constructor, and assignment operator are identical to those shown above in the "Basic Hashmap Implementation" section.

`size()` 和 `empty()` 的实现很容易，因为 `hashmap` 实现会在 `mSize` 数据成员中跟踪记录其大小。需要注意，`size()` 方法返回一个 `size_type`，作为一个返回类型，必须用显式 `hashmap<Key, T, Compare, Hash>` 类型名限定：

```

template <typename Key, typename T, typename Compare, typename Hash>
bool hashmap<Key, T, Compare, Hash>::empty() const
{
    return (mSize == 0);
}

template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::size_type
hashmap<Key, T, Compare, Hash>::size() const
{
    return (mSize);
}

```

`max_size()` 要难一些。乍一看，你可能认为 `hashmap` 容器的最大大小就是所有 `list` 的最大大小之和。不过，在最坏情况下，所有元素都可能散列到同一个桶中。因此，它能支持的最大大小只是一个 `list` 的最大大小。

```

template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::size_type
hashmap<Key, T, Compare, Hash>::max_size() const
{
    // In the worst case, all the elements hash to the

```



```

// same list, so the max_size is the max_size of a single
// list. This code assumes that all the lists have the same
// max_size.
return ((*mElems)[0].max_size());
}

```

最后，swap()的实现只是使用 swap() 工具函数来交换各个数据成员（共4个）。需要注意，vector指针会交换，这是一个常量时间操作。

```

// Just swap the four data members.
// Use the generic swap template.
template <typename Key, typename T, typename Compare, typename Hash>
void hashmap<Key, T, Compare, Hash>::swap(
    hashmap<Key, T, Compare, Hash>& hashIn)
{
    // Explicitly qualify with std:: so the compiler doesn't think
    // it's a recursive call.
    std::swap(*this, hashIn);
}

```

### 编写一个迭代器

最重要的容器需求就是迭代器需求。为了用于通用算法，每个容器都必须提供一个迭代器来访问容器中的元素。迭代器一般应当是一个看上去像是智能指针的类，它要提供重载 operator\* 和 operator->，另外根据其特定行为还要提供其他一些操作。只要迭代器可以提供基本的迭代操作，应该就能一切正常了。

关于迭代器，首先要决定这将是一个什么类型的迭代器，是前向、双向还是随机访问迭代器。随机访问迭代器对于关联容器没有什么意义，因此 hashmap 迭代器作为双向迭代器似乎是一个合理的选择。这意味着，必须提供第 21 章所述的额外操作，包括 operator++、operator--、operator== 和 operator!=。

其次要决定如何对容器中的元素排序。hashmap 是无序的，因此按有序顺序进行迭代可能太难了。无需这样做，你的迭代器可以逐个处理桶，先从第一个桶的元素开始，逐步处理到最后一个桶中的元素。这个顺序对于客户来说可能看上去是随机的，但这是一致而且可重复的。

第三个决定是确定在内部如何表示你的迭代器。实现往往相当依赖于容器的内部实现。迭代器的首要用途就是指示容器中的一个元素。对于 hashmap，每个元素都在一个 STL 列表中，因此 hashmap 迭代器可以是一个包装器，来包装指示当前元素的一个 list 迭代器。不过，双向迭代器的第二个用途是允许客户前进到从当前元素算起的下一个元素或前一个元素。为了从一个桶前进到下一个桶，还需要跟踪当前桶和迭代器所指示的 hashmap 对象。

选择了实现后，必须确定末尾迭代器的一个一致表示。应该记得，末尾迭代器实际上应当是“越过最后一个元素”的标记：对指示容器中最后一个元素的迭代器应用 operator++ 就可以得到这个末尾迭代器。hashmap 迭代器会简单地把 hashmap 中最后一个桶的 list 的末尾迭代器用作自己的末尾迭代器。

### HashIterator 类

根据“编写一个迭代器”小节做的决定，下面该定义 HashIterator 类了。需要说明的第一个问题是，每个 HashIterator 对象都是 hashmap 类的一个特定实例化的迭代器。为了提供这种一对一的映射，HashIterator 还必须是一个类模板，要针对 hashmap 类的同样参数模板化。

类定义中的主要问题是，如何遵循双向迭代器需求。应该记得，所有表现得像是迭代器的东西

都是迭代器。你的类不一定非得派生另一个类才能算是双向迭代器。不过，如果希望自己的迭代器可以用于通用算法函数，就必须指定其特性。前面讨论如何编写 STL 算法时谈到，`iterator_traits` 是一个类模板，它为各个迭代器类型定义了 5 个 typedef。如果愿意，可以为你的新迭代器类型对其部分特殊化。还有一种做法，`iterator_traits` 类模板的默认实现只是把这 5 个 typedef 从迭代器类本身当中取出。因此，可以直接在你的迭代器类中定义这些 typedef。实际上，C++ 使这个工作更为容易。并不需要自行定义，只需派生 `iterator` 类模板，它会提供 typedef。这样一来，只需指定迭代器类型和元素类型作为模板参数提供给 `iterator` 类模板。`HashIterator` 是一个双向迭代器，所以可以指定 `bidirectional_iterator_tag` 作为迭代器类型。其他合法的迭代器类型包括：`input_iterator_tag`、`output_iterator_tag`、`forward_iterator_tag` 和 `random_access_iterator_tag`。元素类型就是 `pair<const Key, T>`。

基本说来，一切证明应当从通用 `iterator` 类模板派生你的迭代器类。

以下是基本 `HashIterator` 类定义：

```
// HashIterator class definition
template<typename Key, typename T, typename Compare, typename Hash>
class HashIterator : public std::iterator<std::bidirectional_iterator_tag,
    pair<const Key, T> >
{
public:
    HashIterator(); // Bidirectional iterators must supply default ctors.
    HashIterator(int bucket,
        typename list<pair<const Key, T> >::iterator listIt,
        const hashmap<Key, T, Compare, Hash>* inHashmap);

    pair<const Key, T>& operator*() const;

    // Return type must be something to which -> can be applied.
    // Return a pointer to a pair<const Key, T>, to which the compiler will
    // apply -> again.
    pair<const Key, T>* operator->() const;
    HashIterator<Key, T, Compare, Hash>& operator++();
    const HashIterator<Key, T, Compare, Hash> operator++(int);

    HashIterator<Key, T, Compare, Hash>& operator--();
    const HashIterator<Key, T, Compare, Hash> operator--(int);

    // Don't need to define a copy constructor or operator= because the
    // default behavior is what we want

    // Don't need destructor because the default behavior
    // (not deleting mHashmap) is what we want.

    // These are ok as member functions because we don't support
    // comparisons of different types to this one.
    bool operator==(const HashIterator& rhs) const;
    bool operator!=(const HashIterator& rhs) const;
protected:
    int mBucket;
    typename list<pair<const Key, T> >::iterator mIt;
    const hashmap<Key, T, Compare, Hash>* mHashmap;

    // Helper methods for operator++ and operator--
```

```

    void increment();
    void decrement();
};

```

如果不明白重载操作符的定义和实现（见后面的“HashIterator 方法实现”小节），可以参考第 16 章了解有关操作符重载的详细内容。

#### HashIterator 方法实现

HashIterator 构造函数会初始化 3 个成员变量。默认构造函数的存在只是为了让客户能够声明 HashIterator 变量而不用对其初始化。用默认构造函数构造的迭代器不必指示任何值，试图对这个迭代器完成操作时，可能有未定义的结果。

```

// Dereferencing or incrementing an iterator constructed with the
// default ctor is undefined, so it doesn't matter what values we give
// here.
template<typename Key, typename T, typename Compare, typename Hash>
HashIterator<Key, T, Compare, Hash>::HashIterator()
{
    mBucket = -1;
    mIt = list<pair<const Key, T> >::iterator();
    mHashMap = NULL;
}

template<typename Key, typename T, typename Compare, typename Hash>
HashIterator<Key, T, Compare, Hash>::HashIterator(
    int bucket, typename list<pair<const Key, T> >::iterator listIt,
    const hashmap<Key, T, Compare, Hash>* inHashMap) :
    mBucket(bucket), mIt(listIt), mHashMap(inHashMap)
{
}

```

解除引用操作符的实现很简洁，但是可能比较难。在第 16 章曾经指出，operator\* 和 operator-> 是非对称的。operator\* 返回实际的底层值，在这里则是迭代器所指示的元素。另一方面，operator-> 所返回的结果必须保证还能再次对其应用箭头操作符。因此，它会返回元素的一个指针。编译器再对该指针应用->，这就会导致访问元素的一个字段。

```

// Return the actual element
template<typename Key, typename T, typename Compare, typename Hash>
pair<const Key, T>& HashIterator<Key, T, Compare, Hash>::operator*() const
{
    return (*mIt);
}

// Return the iterator, so the compiler can apply -> to it to access
// the actual desired field.
template<typename Key, typename T, typename Compare, typename Hash>
pair<const Key, T>*
HashIterator<Key, T, Compare, Hash>::operator->() const
{
    return (&(*mIt));
}

```

自增和自减操作符的实现如第 16 章所述，只不过真正的自增和自减过程是在 increment() 和 decrement() 辅助方法中完成的。

```

// Defer the details to the increment() helper.
template<typename Key, typename T, typename Compare, typename Hash>
HashIterator<Key, T, Compare, Hash>&
    HashIterator<Key, T, Compare, Hash>::operator++()
{
    increment();
    return (*this);
}

// Defer the details to the increment() helper.
template<typename Key, typename T, typename Compare, typename Hash>
const HashIterator<Key, T, Compare, Hash>
    HashIterator<Key, T, Compare, Hash>::operator++(int)
{
    HashIterator<Key, T, Compare, Hash> oldIt = *this;
    increment();
    return (oldIt);
}

// Defer the details to the decrement() helper.
template<typename Key, typename T, typename Compare, typename Hash>
HashIterator<Key, T, Compare, Hash>&
    HashIterator<Key, T, Compare, Hash>::operator--()
{
    decrement();
    return (*this);
}

// Defer the details to the decrement() helper.
template<typename Key, typename T, typename Compare, typename Hash>
const HashIterator<Key, T, Compare, Hash>
    HashIterator<Key, T, Compare, Hash>::operator--(int)
{
    HashIterator<Key, T, Compare, Hash> newIt = *this;
    decrement();
    return (newIt);
}

```

对一个 HashIterator 自增会让它指示容器中的“下一个”元素。这个方法首先让 list 迭代器自增，然后检查是否到达桶的末尾。倘若如此，就寻找 hashmap 中的下一个非空桶，然后设置 list 迭代器等于这个桶中的开始元素。需要注意，它并不能简单地移至下一个桶，因为下一个桶中可能没有任何元素。如果没有更多的非空桶，mIt 会设置为 hashmap 中最后一个桶的末尾迭代器，这是 HashIterator 的特殊“末尾”位置。应该记得，迭代器不必比哑指针更安全，所以对于已经处在末尾的迭代器再自增等情况不必进行错误检查。

```

// Behavior is undefined if mIt already refers to the past-the-end
// element in the table, or is otherwise invalid.
template<typename Key, typename T, typename Compare, typename Hash>
void HashIterator<Key, T, Compare, Hash>::increment()
{
    // mIt is an iterator into a single bucket.
    // Increment it.
    ++mIt;

    // If we're at the end of the current bucket,

```



```

// find the next bucket with elements.
if (mIt == (*mHashMap->mElems)[mBucket].end()) {
    for (int i = mBucket + 1; i < (*mHashMap->mElems).size(); i++) {
        if (!((*mHashMap->mElems)[i].empty())) {
            // We found a nonempty bucket.
            // Make mIt refer to the first element in it.
            mIt = (*mHashMap->mElems)[i].begin();
            mBucket = i;
            return;
        }
    }
    // No more empty buckets. Assign mIt to refer to the end
    // iterator of the last list.
    mBucket = (*mHashMap->mElems).size() - 1;
    mIt = (*mHashMap->mElems)[mBucket].end();
}
}

```

自减是自增的逆操作，它使迭代器指示容器中的“前一个”元素。不过，在此存在非对称性，因为开始和末尾位置就不对称，开始位置是第一个元素，末尾则是最后一个元素之后的位置。自减的算法会首先检查底层 list 迭代器是否是当前桶的开始迭代器。如果不是，就可以自减。否则，代码要检查当前桶之前的第一个非空桶。如果找到了这样一个非空桶，list 迭代器必须设置为指示桶中的最后一个元素，即对末尾迭代器减 1。如果没有找到非空桶，自减就是非法的，所以代码会“肆意妄为”（其行为未定义）。

```

// Behavior is undefined if mIt already refers to the first element
// in the table, or is otherwise invalid.
template<typename Key, typename T, typename Compare, typename Hash>
void HashIterator<Key, T, Compare, Hash>::decrement()
{
    // mIt is an iterator into a single bucket.
    // If it's at the beginning of the current bucket, don't decrement it.
    // Instead, try to find a nonempty bucket ahead of the current one.
    if (mIt == (*mHashMap->mElems)[mBucket].begin()) {
        for (int i = mBucket - 1; i >= 0; --i) {
            if (!((*mHashMap->mElems)[i].empty())) {
                mIt = (*mHashMap->mElems)[i].end();
                --mIt;
                mBucket = i;
                return;
            }
        }
        // No more nonempty buckets. This is an invalid decrement.
        // Assign mIt to refer to one before the start element of the first
        // list (an invalid position).
        mIt = (*mHashMap->mElems)[0].begin();
        --mIt;
        mBucket = 0;
    } else {
        // We're not at the beginning of the bucket, so
        // just move down.
        --mIt;
    }
}

```

要注意 `increment()` 和 `decrement()` 都访问了 `hashmap` 类的 `protected` 成员。因此, `hashmap` 类必须声明 `HashIterator` 是一个 `friend` 类。

分析了 `increment()` 和 `decrement()` 之后, `operator==` 和 `operator!=` 就相对简单了。它们只是完成对象中各个数据成员 (共 3 个) 的比较。

```
template<typename Key, typename T, typename Compare, typename Hash>
bool HashIterator<Key, T, Compare, Hash>::operator==(
    const HashIterator& rhs) const
{
    // All fields, including the hashmap to which the iterators refer,
    // must be equal.
    return (mHashmap == rhs.mHashmap && mBucket == rhs.mBucket &&
        mIt == rhs.mIt);
}

template<typename Key, typename T, typename Compare, typename Hash>
bool HashIterator<Key, T, Compare, Hash>::operator!=(
    const HashIterator& rhs) const
{
    return (!operator==(rhs));
}
```

#### const 迭代器

理论上讲, 应当为 `hashmap` 类同时提供一个迭代器和一个 `const` 迭代器。`const` 迭代器的功能类似于迭代器, 但是应当对元素提供只读访问。迭代器总是可以转换为一个 `const` 迭代器。有关 `const` 迭代器的详细内容我们不做介绍, 其实现作为练习留给读者来完成。

#### 迭代器 typedef 和访问方法

为 `hashmap` 提供迭代器支持的最后一个工作是在 `hashmap` 类定义中提供必要的 typedef, 并编写 `hashmap` 的 `begin()` 和 `end()` 方法。typedef 和方法原型如下所示。

```
template <typename Key, typename T, typename Compare = std::equal_to<Key>,
    typename Hash = DefaultHash<Key> >
class hashmap
{
public:
    // Other typedefs omitted for brevity
    typedef HashIterator<Key, T, Compare, Hash> iterator;
    typedef HashIterator<Key, T, Compare, Hash> const_iterator;

    // Iterator methods
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
    // Remainder of class definition omitted for brevity
};
```

`begin()` 最难的部分是要记住, 如果散列表中没有任何元素就要返回末尾迭代器。

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::iterator
hashmap<Key, T, Compare, Hash>::begin()
{
```



```

    if (mSize == 0) {
        // Special case: there are no elements, so return the end iterator
        return (end());
    }

    // We know there is at least one element. Find the first element.
    for (size_t i = 0; i < mElems->size(); ++i) {
        if (!((*mElems)[i].empty())) {
            return (HashIterator<Key, T, Compare, Hash>(i,
                (*mElems)[i].begin(), this));
        }
    }
    // Should never reach here, but if we do, return the end iterator
    return (end());
}

```

end()会创建 HashIterator 指示最后一个桶的末尾迭代器。

```

template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::iterator
hashmap<Key, T, Compare, Hash>::end()
{
    // The end iterator is just the end iterator of the list in last bucket.
    return (HashIterator<Key, T, Compare, Hash>(mElems->size() - 1,
        (*mElems)[mElems->size() - 1].end(), this));
}

```

由于我们没有提供一个 const\_iterator, const 版本的 begin()和 end()与非 const 版本的 begin()和 end()是相同的。

#### 使用 HashIterator

现在既然 hashmap 支持迭代, 所以可以像在任何 STL 容器上一样对其元素进行迭代处理, 而且可以将迭代器传递给方法和函数。

```

#include "hashmap.h"
#include <iostream>
#include <map>
using namespace std;

int main(int argc, char** argv)
{
    hashmap<string, int> myHash;
    myHash.insert(make_pair("KeyOne", 100));
    myHash.insert(make_pair("KeyTwo", 200));
    myHash.insert(make_pair("KeyThree", 300));

    for (hashmap<string, int>::iterator it = myHash.begin();
        it != myHash.end(); ++it) {
        // Use both -> and * to test the operations.
        cout << it->first << " maps to " << (*it).second << endl;
    }

    // Create a map with all the elements in the hashmap.
    map<string, int> myMap(myHash.begin(), myHash.end());
    for (map<string, int>::iterator it = myMap.begin();

```

```

        it != myMap.end(); ++it) {
            // Use both -> and * to test the operations.
            cout << it->first << " maps to " << (*it).second << endl;
        }

        return (0);
    }

```

### 有关分配器的说明

本章较早前已经说过，所有 STL 容器都允许指定一个定制内存分配器。一个好的 hashmap 实现也应当如此。不过，这些细节在此并不给出，因为这会混淆视听，使我们反而忽略实现的重点。

### 有关可逆容器的说明

如果你的容器提供了一个双向迭代器或随机访问迭代器，则认为是可逆的。可逆容器要提供另外两个 typedef，如表 23-3 所示。

表 23-3

类 型 名	说 明
reverse_iterator	“智能指针”的类型，以逆序对容器中的元素迭代处理
const_reverse_iterator	reverse_iterator 的另一个版本，以逆序对容器中的元素迭代处理

另外，容器还应提供 rbegin() 和 rend()，它们分别与 begin() 和 end() 对称（译者注：并不是说 rbegin() 和 rend() 对称，而是指 rbegin() 和 begin() 对称或对应，rend() 与 end() 对称或对应）。通常的实现只是使用本章前面所述的 reverse\_iterator 适配器。其实现作为练习留给读者来完成。

### hashmap 作为关联容器

除了已经看到的基本容器需求，可以让你的容器遵循关联容器或顺序容器的额外需求。hashmap 类似于 map，显然是一个关联容器，因此要提供以下 typedef 和方法。

### 关联容器的 typedef 需求

关联容器需要另外 3 个 typedef，如表 23-4 所示。

表 23-4

类 型 名	说 明
key_type	键类型，容器利用此键类型实例化
key_compare	比较类或函数指针类型，容器利用此类型实例化
value_compare	比较两个 value_type 元素的类

我们的实现还包括一个 mapped\_type typedef，因为 map 就提供了这样一个 typedef。value\_compare 并没有实现为一个 typedef，而是作为一个嵌套类定义。也可以采用另外一种做法，此类可以是 hashmap 的一个 friend 类，但这个定义要遵循标准中所给出的 map 定义。value\_compare 类的用途是对两个元素的键调用比较函数。

```

template <typename Key, typename T, typename Compare = std::equal_to<Key>,
          typename Hash = DefaultHash<Key> >
class hashmap
{

```

```
public:
```

```
    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
    typedef Compare key_compare;
    typedef pair<const Key, T>& reference;
    typedef const pair<const Key, T>& const_reference;
    typedef HashIterator<Key, T, Compare, Hash> iterator;
    typedef HashIterator<Key, T, Compare, Hash> const_iterator;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
```

```
// Required class definition for associative containers
```

```
class value_compare :
```

```
public std::binary_function<value_type, value_type, bool>
```

```
{
```

```
    friend class hashmap<Key, T, Compare, Hash>;
```

```
public:
```

```
    bool operator() (const value_type& x, const value_type& y) const
```

```
{
```

```
        return comp(x.first, y.first);
```

```
}
```

```
protected:
```

```
    Compare comp;
```

```
    value_compare(Compare c) : comp(c) {}
```

```
};
```

```
// Remainder of hashmap class definition omitted for brevity
```

```
};
```

### 关联容器的方法需求

标准对于关联容器预先指定了一些额外的方法需求，如表 23-5 所示。

表 23-5

方 法	说 明	复 杂 性
取一个迭代器区间的构造函数	构造容器，并插入迭代器区间中的元素。迭代器区间不一定指示另一个同类型的容器。需要注意，关联容器的所有构造函数都必须取一个类型为 value_compare 的比较对象。构造函数应当提供一个默认的构造对象作为默认值	$N \log N$
key_compare key_comp() const; value_compare value_comp() const;	返回比较对象来比较键或者比较整个值	常量时间
pair<iterator, bool> insert (value_type&); iterator insert (iterator, value_type&); void insert (InputIterator start, InputIterator end);	三种形式的插入。第二个插入方法中的 iterator 位置是一个提示，此提示可以忽略。第三个插入方法中的区间不一定是一个同类型容器中的区间。允许有重复键的容器只从第一种形式返回 iterator，因为 insert() 总会成功	对数时间

(续)

方 法	说 明	复 杂 性
<pre>size_type erase (key_type&amp;); void erase (iterator); void erase (iterator start, iterator end);</pre>	三种形式的删除。第一种形式返回所删除值的个数 (对于不允许重复键的容器, 只会返回 0 或 1)。第二种形式和第三种形式会删除 iterator 位置上的元素, 或者删除从 start 到 end 的区间内的元素	对数时间, 但第二种形式除外, 这种形式的删除应当是摊分常量时间
<pre>void clear();</pre>	删除所有元素	线性时间
<pre>iterator find (key_type&amp;); const_iterator find (key_type&amp;) const;</pre>	找到有指定键的元素	对数时间
<pre>size_type count (key_type&amp;) const;</pre>	返回有指定键的元素的个数 (对于不允许重复键的容器只返回 0 或 1)	对数时间
<pre>iterator lower_bound (key_type&amp;); iterator upper_bound (key_type&amp;); pair&lt;iterator, iterator&gt; equal_range (key_type&amp;); const_iterator lower_bound (key_type&amp;) const; const_iterator upper_bound (key_type&amp;) const; pair&lt;const_iterator, const_iterator&gt; equal_range (key_type&amp;) const;</pre>	返回指示有指定键的第一个元素的迭代器, 或指示有指定键的最后一个元素之后位置的迭代器, 或者二者都返回	对数时间

要注意 lower\_bound()、upper\_bound() 和 equal\_range() 只对有序容器有意义。因此 hashmap 没有提供这些方法。

以下是完整的 hashmap 类定义。需要指出, 与前面介绍的 insert()、erase() 和 find() 相比, 这些方法的原型稍有修改。

```
template <typename Key, typename T, typename Compare = std::equal_to<Key>,
typename Hash = DefaultHash<Key> >
class hashmap
{
public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
```

```

typedef Compare key_compare;
typedef pair<const Key, T>& reference;
typedef const pair<const Key, T>& const_reference;
typedef HashIterator<Key, T, Compare, Hash> iterator;
typedef HashIterator<Key, T, Compare, Hash> const_iterator;
typedef size_t size_type;
typedef ptrdiff_t difference_type;
// Required class definition for associative containers
class value_compare :
    public std::binary_function<value_type, value_type, bool>
{
    friend class hashmap<Key, T, Compare, Hash>;
public:
    bool operator() (const value_type& x, const value_type& y) const
    {
        return comp(x.first, y.first);
    }
protected:
    Compare comp;
    value_compare(Compare c) : comp(c) {}
};

```

```

// The iterator class needs access to protected members of the hashmap.
friend class HashIterator<Key, T, Compare, Hash>;

```

```

// Constructors
explicit hashmap(const Compare& comp = Compare(),
    const Hash& hash = Hash()) throw(invalid_argument);

```

```

template <class InputIterator>
hashmap(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Hash& hash = Hash())
    throw(invalid_argument);

```

```

// destructor, copy constructor, assignment operator
~hashmap();
hashmap(const hashmap<Key, T, Compare, Hash>& src);
hashmap<Key, T, Compare, Hash>& operator=(
    const hashmap<Key, T, Compare, Hash>& rhs);

```

```

// Iterator methods
iterator begin();
iterator end();
const_iterator begin() const;
const_iterator end() const;

```

```

// Size methods
bool empty() const;
size_type size() const;
size_type max_size() const;

```

```

// Element insert methods
T& operator[] (const key_type& x);
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator position, const value_type& x);

```

```

template <class InputIterator>
void insert(InputIterator first, InputIterator last);

// Element delete methods
void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);

// Other modifying utilities
void swap(hashmap<Key, T, Compare, Hash>& hashIn);
void clear();

// Access methods for STL conformity
key_compare key_comp() const;
value_compare value_comp() const;

// Lookup methods
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
size_type count(const key_type& x) const;

```

```

protected:
    typedef list<value_type> ListType;

    typename ListType::iterator findElement(
        const key_type& x, int& bucket) const;
    vector<ListType>* mElems;
    size_type mSize;
    Compare mComp;
    Hash mHash;
};

```

### hashmap 构造函数

默认构造函数的实现前面已经给出了。第二个构造函数是一个方法模板，因此它可以取任何容器的一个迭代器区间作为参数，而不只是其他 hashmap 的迭代器区间。如果不是一个方法模板，就需要显式地指定 InputIterator 类型为 HashIterator，以此限制它是 hashmap 的迭代器。尽管语法有点复杂，但实现并不复杂，它会初始化所有数据成员，然后调用 insert() 来具体插入指定区间中的所有元素。

```

// Make a call to insert() to actually insert the elements.
template <typename Key, typename T, typename Compare, typename Hash>
template <class InputIterator>
hashmap<Key, T, Compare, Hash>::hashmap(
    InputIterator first, InputIterator last, const Compare& comp,
    const Hash& hash) throw(invalid_argument) : mSize(0), mComp(comp), mHash(hash)
{
    if (mHash.numBuckets() <= 0) {
        throw (invalid_argument("Number of buckets must be positive"));
    }
    mElems = new vector<list<value_type> >(mHash.numBuckets());
    insert(first, last);
}

```

### hashmap 插入操作

第一个版本的 insert() 向 hashmap 中增加一个键/值对。这与“基本 hashmap”小节中所示的版本基





学习在线

视频资料下载  
电子书交流

[www.eimhe.com](http://www.eimhe.com)

本相同，只不过在此要返回一个 iterator/bool pair。iterator 必须是一个 HashIterator，这个对象构造为指示刚刚插入的元素，或者如果已经存在指定键的元素，则指示这个已经存在的元素。

```
template <typename Key, typename T, typename Compare, typename Hash>
pair<typename hashmap<Key, T, Compare, Hash>::iterator, bool>
hashmap<Key, T, Compare, Hash>::insert(const value_type& x)
{
    int bucket;
    // Try to find the element.
    typename ListType::iterator it = findElement(x.first, bucket);

    if (it != (*mElems)[bucket].end()) {
        // The element already exists.
        // Convert the list iterator into a HashIterator, which
        // also requires the bucket and a pointer to the hashmap.
        HashIterator<Key, T, Compare, Hash> newIt(bucket, it, this);

        // Some compilers don't like make_pair here.
        pair<HashIterator<Key, T, Compare, Hash>, bool> p(newIt, false);
        return (p);
    } else {
        // We didn't find the element, so insert a new one.
        mSize++;
        typename ListType::iterator endIt =
            (*mElems)[bucket].insert((*mElems)[bucket].end(), x);
        pair<HashIterator<Key, T, Compare, Hash>, bool> p(
            HashIterator<Key, T, Compare, Hash>(bucket, endIt, this), true);
        return (p);
    }
}
```

取一个位置的 insert() 版本对于 hashmap 并没有用。实现完全忽略了 position，这就退化为第一个版本的 insert()。

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::iterator
hashmap<Key, T, Compare, Hash>::insert(typename hashmap<Key, T, Compare,
Hash>::iterator position, const value_type& x)
{
    // Completely ignore position
    return (insert(x).first);
}
```

第三种形式的 insert() 是一个方法模板，其原因与前面所示的构造函数相同，它应当能够使用任何类型容器的迭代器插入元素。具体实现使用了一个 insert\_iterator，这在本章前面介绍过。

```
template <typename Key, typename T, typename Compare, typename Hash>
template <class InputIterator>
void hashmap<Key, T, Compare, Hash>::insert(InputIterator first,
InputIterator last)
{
    // Copy each element in the range by using an insert_iterator
    // adapter. Give begin() as a dummy position--insert ignores it
    // anyway.
    insert_iterator<hashmap<Key, T, Compare, Hash> > inserter(*this, begin());
    copy(first, last, inserter);
}
```

### hashmap 删除操作

第一个版本的 `erase()` 与“基本 Hashmap”一节中所示的版本基本相同，只不过在此会返回所删除元素的个数（0 或 1）。

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::size_type
hashmap<Key, T, Compare, Hash>::erase(const key_type& x)
{
    int bucket;

    // First, try to find the element.
    typename ListType::iterator it = findElement(x, bucket);

    if (it != (*mElems)[bucket].end()) {
        // The element already exists--erase it.
        (*mElems)[bucket].erase(it);
        mSize--;
        return (1);
    } else {
        return (0);
    }
}
```

第二个版本的 `erase()` 必须删除特定迭代器位置上的元素。当然，给定的迭代器是一个 `HashIterator`。因此，`hashmap` 必须能够从 `HashIterator` 得到底层桶和 `list` 迭代器。我们采用的方法是让 `hashmap` 类作为 `HashIterator` 的一个友元（在前面的类定义中没有显示）。

```
template <typename Key, typename T, typename Compare, typename Hash>
void hashmap<Key, T, Compare, Hash>::erase(
    hashmap<Key, T, Compare, Hash>::iterator position)
{
    // Erase the element from its bucket.
    (*mElems)[position.mBucket].erase(position.mIt);
    mSize--;
}
```

最后一个版本的 `erase()` 会删除一个元素区间。它只是从 `first` 到 `last` 迭代处理，对每个元素调用 `erase()`，所以这是利用前一个版本的 `erase()` 来完成所有工作。

```
template <typename Key, typename T, typename Compare, typename Hash>
void hashmap<Key, T, Compare, Hash>::erase(
    hashmap<Key, T, Compare, Hash>::iterator first,
    hashmap<Key, T, Compare, Hash>::iterator last)
{
    typename hashmap<Key, T, Compare, Hash>::iterator cur, next;

    // Erase all the elements in the range.
    for (next = first; next != last; ) {
        cur = next++;
        erase(cur);
    }
}
```

`clear()` 使用 `for_each()` 算法对表示各个桶的 `list` 调用 `clear()`。

```
template <typename Key, typename T, typename Compare, typename Hash>
void hashmap<Key, T, Compare, Hash>::clear()
{
    // Call clear on each list.
    for_each(mElems->begin(), mElems->end(), mem_fun_ref(&ListType::clear));
    mSize = 0;
}
```

hashmap 访问操作

标准要求为键比较和值比较对象提供访问方法。

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::key_compare
hashmap<Key, T, Compare, Hash>::key_comp() const
{
    return (mComp);
}

template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::value_compare
hashmap<Key, T, Compare, Hash>::value_comp() const
{
    return (value_compare(mComp));
}
```

find()方法与基本 hashamp 的相应版本基本相同，只是返回码有所不同。它并不返回元素的一个指针，而是会构造一个指示该元素的 HashIterator。const 版本与之相同，所以这里没有给出 const 版本的实现。

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::iterator
hashmap<Key, T, Compare, Hash>::find(const key_type& x)
{
    int bucket;
    // Use the findElement() helper.
    typename ListType::iterator it = findElement(x, bucket);
    if (it == (*mElems)[bucket].end()) {
        // We didn't find the element--return the end iterator.
        return (end());
    }
    // We found the element--convert the bucket/iterator to a HashIterator.
    return (HashIterator<Key, T, Compare, Hash>(bucket, it, this));
}
```

count()的实现是 find()的一个包装器，如果找到了元素会返回 1，否则返回 0。应该记得，如果 find()没有找到元素，它会返回末尾迭代器。count()通过调用 end()来获取一个末尾迭代器，以便进行比较。

```
template <typename Key, typename T, typename Compare, typename Hash>
typename hashmap<Key, T, Compare, Hash>::size_type
hashmap<Key, T, Compare, Hash>::count(const key_type& x) const
{
    // There are either 1 or 0 elements matching key x.
    // If we can find a match, return 1, otherwise return 0.
}
```

```
    if (find(x) == end()) {  
        return (0);  
    } else {  
        return (1);  
    }  
}
```

最后一个方法在标准中未做要求，不过为了使用方便，在此提供了这个方法。其原型和实现等同于 STL map 中 operator [] 的原型和实现。这个实现只有一行，很容易让人糊涂，所以在此提供了注释来加以解释。

```
template <typename Key, typename T, typename Compare, typename Hash>  
T& hashmap<Key, T, Compare, Hash>::operator[] (const key_type& x)  
{  
    // This definition is the same as that used by map, according to  
    // the standard.  
    // It's a bit cryptic, but it basically attempts to insert  
    // a new key/value pair of x and a new value. Regardless of whether  
    // the insert succeeds or fails, insert() returns a pair of an  
    // iterator/bool. The iterator refers to a key/value pair, the  
    // second element of which is the value we want to return.  
    return (((insert(make_pair(x, T()))).first)->second);  
}
```

#### 有关顺序容器的说明

前几节开发的 hashmap 是一个关联容器。不过，也可以编写一个顺序容器，在这种情况下，需要遵循另外一组需求。这里没有列出顺序容器的需求，可以指出，deque 容器几乎完全遵循预先指定的顺序容器需求，所以更容易的理解是可以比对着 deque 容器来实现顺序容器。惟一的区别是 deque 容器还提供了一个额外的 resize() 方法（标准中不要求提供这个方法），有关 deque 功能的详细内容请参见本书网站上的标准库参考资源。

## 23.4 小结

本章最后的例子几乎展示了开发一个 hashmap 关联容器及其迭代器的全过程。这个 hashmap 实现与其他例子都可以从网站上下载得到，有关如何下载等内容请见“引言”中的说明。完全可以把它纳入到你的程序中。在读这一章的过程中，你可能还会对如何开发容器的步骤有所认识。即使你不会再写其他 STL 算法或容器，通过本章的学习，也能够更好地理解 STL 的作用和功能，而且可以更好地加以使用。

本章是 STL 之旅的最后一程。虽然我们用了三章来介绍，但是还有一些特性没有谈到，如果你对这部分内容感兴趣，可以参考附录 B 所列的资料来了解更多的信息。另一方面，我们意识到，这一章的语法和内容有些复杂。正如第 21 章和第 22 章所述，不要认为非得使用这里讨论的所有特性。如果没有真正的需要，强制性地地在程序中纳入这些特性，只会让程序更复杂。不过，我们建议，在合适的情况下，应当在程序中结合 STL 的某些方面。先从容器开始，可能提供一个或两个算法，在不知不觉中，你可能就已经改头换面了！



## 第 24 章 探讨分布式对象

分布式计算是这样一种思想，就是将程序的操作分布在网络上多台计算机上完成。随着网络变得越来越快，越来越常用，越来越多的应用会在处理时利用到网络上的其他计算机。

在本章中，你将了解到分布式对象（distributed object）的概念：就是应用面向对象技术来完成分布式计算。本章先更为详细地定义了分布式计算和分布式对象，还提供了一些示例用例。接下来，将学习 CORBA，这是一种实现分布式对象编程的功能强大的体系结构。最后，还将了解到 XML 技术及其在分布式计算中的地位。

### 24.1 分布式计算的魅力

最近 10 年内，随着 Internet 日渐占据突出的地位，分布式计算也得到了越来越多的关注。不过这不仅仅是一句时髦的口号，对于某些类型的应用来说，分布式程序是再合适不过的了。例如，可以想像一下要编写一个程序来包含 Web 上的所有可用信息，这几乎是不可能的。只有 Web 才有能力包含大量数据和动态内容，因为它是分布在许多不同的机器上的。

#### 24.1.1 分布以获得可扩展性

从每一天的来看，桌面计算机可能在大部分时间内都没有做任何事情。即使你很积极地使用计算机，但是现代处理器速度太快，所以在许多时间内处理器都是空闲的，等着人跟上它的速度。事实上，当今世界上有许许多多的计算机，其中大多数都未超负荷工作，不过这些计算机时刻都准备投入使用。大多数计算机用户都不能充分利用桌面 PC 机的能力。如果使用一个监控工具来检查处理器的利用率，就会发现在任何时间内利用率都很少达到 100%。不过，有些应用非常耗费处理器的处理能力。要在桌面计算机上运行一个包括大量复杂计算的程序可能会花费数小时之多。如果把网络上未用处理器的能力充分利用起来，运行这个程序的时间就会大大减少。这种技术称为网格计算（grid computing）。

在利用分布来提高速度的应用当中，一个经典的实际例子就是图像渲染。仅仅生成视频质量计算机动画的一个静止帧就需要大量的计算。3-D 渲染应用多年以来已经大量利用了分布式计算。所谓的渲染场（rendering farm）可以利用高速网络连接来组建。一台机器可以相当于中心“大脑”，它将小块的计算分发给场中的各个机器。作为大脑的这台机器会收集各个机器的计算结果，从而形成最终的图像或视频。

SETI@home 项目也是一个使用分布式计算来提高计算性能的例子。这个程序将来自太空的信号分析工作分布至所有参与的计算机上，以此为“对外星智能的探索”（Search for Extra-Terrestrial Intelligence, SETI）提供辅助。如果在一台计算机上分析如此多的数据，这是不实际的。SETI@home 为用户提供一个程序，用户可以在家用 PC 上运行，这样当这个计算机未做其他使用时就可以处理部分数据块。当然，还不能说这已经取得了成功（我们还没有发现外星球上的小绿人），不过，作为一个早期推广的分布式 Internet 应用，它确实有助于这项技术的普及。



### 24.1.2 分布以获得可靠性

分布式计算可以视为墨菲定律（只要可能出错，就会出错）的一个解决方案。有些应用（如网站和数据库）必须一直可用而且一直处于工作状态。可以把这些应用编写为在网络中的多台计算机上运行。如果一个机器出问题了，另一个机器可以立即接管过去。在诸如此类的情况下，分布主要是一个同步问题。应用的所有实例都必须有足够的通信，这样在出现故障时，用户并不会看到任何变化。

在访问一个大规模网站时，实际上会与多台服务器之一连接，所有服务器都会包含相同数据的镜像。如果一台机器出故障，负载均衡器就不再向其发送任何请求，直到它恢复为止。

### 24.1.3 分布以获得集中性

网络上要是有一个系统来控制 and 监视其他应用的行为，这通常很有用。例如，Sassafras Software 的 KeyServer 就是这样一个应用，它允许网络管理员确保其网络上使用的软件不会违反许可条款。当网络上的一个用户启动某个应用时，这个应用就会“联系”KeyServer，请求运行许可。KeyServer 会跟踪目前运行了多少个副本。如果还有可用许可，发出请求的应用就可以正常启动。顺带地，KeyServer 还利用了分布来获得可靠性，管理员可以安装多个“镜像”KeyServer，以便在某个 KeyServer 出问题的时候能够妥善处理。

### 24.1.4 分布式内容

对等（端到端，Peer-to-peer）应用最近以飞快的速度流行起来。其基本思想是，网络上的用户都运行一个应用，允许他们基于一对一的形式进行通信。在一个文件共享应用中，每个用户都将允许别人通过网络访问存储在本地计算机上的文件。寻找文件的用户可以连接到有此文件的用户，并开始文件传输。通过将文件共享应用分布到所有用户，应用就能提供更多内容，要是仅由单个服务器存放信息，信息量绝对无法与之比拟。这样还能分布通信负载，所以不会出现中心服务器成为所有请求的瓶颈的情况。

### 24.1.5 分布式 vs 网络式

要记住，并非所有利用到网络的应用都一定是分布式应用。网络式应用也会与其他机器通信或传输数据。分布式（distributed）表示一种内容更丰富的交互。

有时很难做出区别。例如，考虑一个视频游戏。如果这个游戏与一个中心服务器通信，来检查更新情况，这就是一个网络式应用，因为游戏本身并不是在多个计算机上运行，它只是在操作过程中与另一台机器通信而已。不过，如果游戏允许多个玩家在不同机器上参与游戏，就可以实现为一个分布式应用。

要明确一个应用究竟是一个分布式应用还是网络式应用（不过，这只是一个学术研究上的区别，并没有真正的影响），最好的办法是从一个机器的角度出发，确定计算是否在另一台机器上发生。在网络式游戏中，游戏的各个副本很可能在自己的机器上操作。它们只是在需要状态更新时才使用网络，如图 24-1 所示。由图可见，当机器 A 上的玩家开枪时，有关此次开枪的信息就会传输至机器 B。这两台机器会基于同一个事件采取适当的动作。这种网络式游戏通常不认为是一个分布式应用。



图 24-1

可以想像这样一个游戏，它确实以分布式方式实现。其中，并不是由一个机器向另一个机器发出一个事件，再独立地处理结果，这个处理只会完成一次，而且是分布在两个机器上完成的。如图 24-2 所示。机器 A 可以发出事件，让机器 B 确定结果是什么。从机器 A 的角度看，有些数据处理是在外部发生的。这就很好地说明了它在运行一个分布式应用。

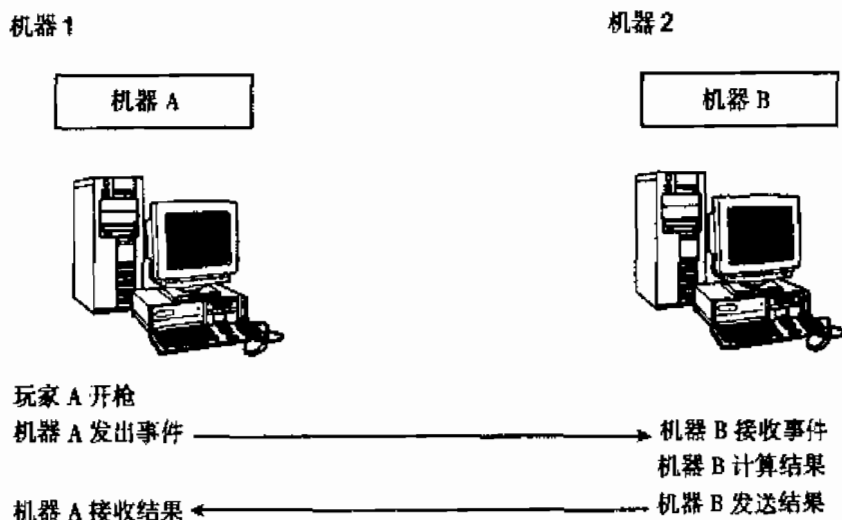


图 24-2

## 24.2 分布式对象

早在面向对象程序设计 (object-oriented programming, OOP) 流行起来之前，分布式计算就已经存在了，所以这两个概念有着显著的区别。不过，OOP 概念应用到分布式计算可以得到极其强大的新抽象。如果想像一下，如果要对某个对象调用一个方法，而这个对象实际上在数千里之外的某台计算机上，或者是要在网络上的不同主机之间传递一个对象，考虑这样一些情况就能了解到 OOP 与分布式计算相结合有着怎样的好处。

### 24.2.1 串行化和编组

基本说来，网络只知道如何传输原始数据，并不知道有关 C++、对象或代码执行的任何内容。网络只会把数据从一个地方移到另一个地方。这种简单性正是网络诸多优秀特性之一。一个网络中可以包括异构的计算机集合，这些计算机可以有不同的体系结构和操作系统，因为网络对网络中组件的环境没有做多少假设。

对于分布式应用，网络的简单性会带来一个小问题。如果能够调用一个函数将对象放在网络上传送，从而把对象从一个机器发送到另一个机器，这应该很不错，不过这比听上去要复杂，因为网络不知道对象的存在。实际上，你需要将对象转换为原始字节。并不是发送实际的对象，而是必须发送描述对象的数据。接收者需要再解释原始字节从而重新构造对象，这往往是原始对象的一个复制。

将内存中的对象转换为一种平面的原始表示，这个过程称为串行化 (serialization) 或编组 (marshaling)。重新构造对象的过程称为逆串行化 (deserialization) 或解组 (unmarshalling)。由第 14 章的介绍，你应该对编组不再陌生，第 14 章提供了一个使用字符串表示对象的例子。编组不光对网络应用有用。如果想把对象保存到磁盘上，往往需要先把它编组为一种平面格式。

#### 实战串行化

考虑以下函数声明，它能够通过网络向另一个计算机发送数据，而且能够接收来自另一个计算机的数

据。如第 18 章所述, C++ 没有提供内置的网络功能, 所以你的操作系统所提供的具体网络库很可能与在此所示的网络库有所不同, 这里的网络库极其简单。

```
/**
 * Sends data to another host on the network
 *
 * @param inHostName the name of the other machine
 * @param inData      the data you want to send
 */
void send(const string& inHostName, const string& inData);

/**
 * Receives incoming data from the network
 *
 * @return the data that was received
 */
string read();
```

假设你在为一家公司编写一个库存控制应用, 这家公司的仓库遍布在美国各地。每个仓库所在位置都会运行程序的一个副本, 但是程序需要能够履行各地的订单。换句话说, 可以使用在纽约 Pittsford 运行的程序来订购位于华盛顿 Onion Creek 的仓库里的一件商品。由于各个位置上的所有程序都使用相同的 Order 类来表示一个订单, 你要做的就是通过网络将订单从 Pittsford 发送到 Onion Creek。以下是 Order 类的类定义:

```
// Order.h

class Order
{
public:
    Order();

    int getItemNumber() const;
    void setItemNumber(int inItemNumber);

    int getQuantity() const;
    void setQuantity(int inQuantity);

    int getCustomerNumber() const;
    void setCustomerNumber(int inCustomerNumber);

protected:
    int mItemNumber;
    int mQuantity;
    int mCustomerNumber;
};
```

在此, 要发送的数据是对象, 但是网络没有能力传送对象 (它只能处理 string), 这就存在着不匹配。解决方案就是在 Order 类中增加串行化和逆串行化功能。这个新的类定义如下所示:

```
// Order.h

#include <string>

class Order
{
```

```

public:
    Order();

    int getItemNumber() const;
    void setItemNumber(int inItemNumber);

    int getQuantity() const;
    void setQuantity(int inQuantity);

    int getCustomerNumber() const;
    void setCustomerNumber(int inCustomerNumber);

```

```

/**
 * Converts the object into raw data that can be sent over the
 * network
 *
 * @return      a string representing this object
 */
std::string serialize();

/**
 * Adjusts this object to represent the data in inData
 *
 * @param inData      a string representing Order data
 */
void deserialize(const std::string& inData);

```

```

protected:
    int mItemNumber;
    int mQuantity;
    int mCustomerNumber;
};

```

Order 类的实现如下所示。这里只突出显示了串行化方法，因为类实现中剩下的部分没有太大意思。

// Order.cpp

```

#include "Order.h"
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

Order::Order() : mItemNumber(-1), mQuantity(-1), mCustomerNumber(-1)
{
}

int Order::getItemNumber()
{
    return mItemNumber;
}

void Order::setItemNumber(int inItemNumber)
{
    mItemNumber = inItemNumber;
}

```

```
int Order::getQuantity()
{
    return mQuantity;
}

void Order::setQuantity(int inQuantity)
{
    mQuantity = inQuantity;
}

int Order::getCustomerNumber()
{
    return mCustomerNumber;
}

void Order::setCustomerNumber(int inCustomerNumber)
{
    mCustomerNumber = inCustomerNumber;
}
```

```
string serialize()
{
    // Use a stream to output all the values, separated by tabs.
    ostringstream outStream;

    outStream << getItemNumber() << "\t" <<
        getQuantity() << "\t" <<
        getCustomerNumber();

    return outStream.str();
}

void deserialize(const string& inData)
{
    // Use an input stream to read the values back in the same order.
    istream inStream(inData);

    if (!inStream.good()) {
        cerr << "Error deserializing!" << endl;
    } else {
        inStream >> mItemNumber;
        inStream >> mQuantity;
        inStream >> mCustomerNumber;
    }
}
```

只需提供一种方法来完成对象和字符串的转换，就可以通过网络传输对象的一种表示。如果程序大量使用了串行化，可能要在每个类中包括 `operator<<` 和 `operator>>` 来提供串行化。

### 24.2.2 远程过程调用

远程过程调用 (remote procedure call, RPC) 是指调用在另一台机器上执行的一个方法或函数的概念行为。在 C++ 中，这个行为本质上讲是概念性的 (conceptual)，因为对于未以某种方式链接至实际程序库的函数，C++ 语言并没有对如何调用这种函数提供任何实际的机制。相反，C++ 中的 RPC 通常会涉及本地的一个桩方法 (stub method)，由它具体隐藏从远程主机获得结果的网络代码。

通过使用操作系统的网络功能和前面所述的串行化技术，就可以编写自己的 RPC 机制了。可以定义

一个类来表示一个远程主机，其中包括一些桩方法，这些方法实际上在远程主机上执行，但是会把结果返回给本地调用者。以下就是这样一个类定义，其中包含的桩方法可以用于得到有关远程机器状态的信息或者可以完成一个重启。

```
class RemoteHost
{
public:
    /**
     * Creates a remote host, which is available at the
     * given address
     */
    RemoteHost(const string& inAddress);

    int getNumConnectedUsers() const;
    int getAvailableMemory() const ;
    int getAvailableDiskSpace() const ;

    void restartNow();

protected:
    string mAddress;
};
```

这些方法的实现仍正常定义，不过可能会利用网络库从远程机器上得到实际的结果。取决于网络库的不同，实际代码也可能有所不同，不过以下显示了一些伪代码，在此利用了 24.2.1 节定义的网络函数：

```
int RemoteHost::getAvailableMemory() const
{
    // Send the string "getAvailableMemory()" to the remote host, instructing
    // it to send back its available memory.
    send(mAddress, "getAvailableMemory()");

    // Get the result from the remote host.
    string result = read();

    // Convert result into an int.
    // Return the int result.
}
```

远程主机上的实现要解析和解释接收到的消息，以使用正确的结果做出响应。下面是远程主机实现的伪代码：

```
void respondToRPC(const string& inRequestHost, const string& inMessage)
{
    string response = "";

    // Look at the message to determine which operation is requested.
    if (inMessage == "getAvailableMemory()") {
        // Use a local function to get the available memory on this machine.
        int memAvail = getAvailMem();
        // Convert the result into a string and put it in the response variable.
    } else if (...) {
        // Handle other messages.
    }
}
```



```
// Send the response back to the requestor.  
send(inRequestHost, response);  
}
```

前面的这个伪代码例子看上去很简单，不过如果想把它转换成实际的产品代码，很快就会出现一些复杂的问题。将遇到的一些问题包括：

- 如何处理不同的数据类型，包括诸如对象等复杂类型。
- 如何处理远程机器不认识的 RPC 调用？
- 如何处理版本化？如果有些远程机器升级了，不再像其他机器那样做出响应怎么办？
- 如何处理网络错误、主机丢失、网络超负荷等等问题？
- 如何处理平台问题，如字节顺序的差别？

出于这些原因，而且一般不想从头来过，程序员通常会使用一个既有的 RPC 包以便实现这种交互。本章余下的部分将介绍 CORBA 和 XML，这是两种完全不同的技术，不过都能用于 RPC 通信。

## 24.3 CORBA

公共对象请求代理体系结构 (Common Object Request Broker Architecture, CORBA) 是一个独立于语言和平台的标准化体系结构，可以用以定义、实现和使用分布式对象。CORBA 的主要目标是提供一个编程环境来隐藏串行化和远程过程调用 (见 24.2.2 节的讨论) 的所有细节。CORBA 还支持位置透明性 (location transparency)：你可以编写使用对象的代码，而无需知道这些对象究竟是本地对象还是远程对象。

CORBA 体系结构本身并不是一个实现，实际上它包括多个标准。其中两个最重要的标准是接口定义语言 (Interface Definition Language, IDL) 和 Internet ORB 间协议 (Internet Inter-ORB Protocol, IIOP)，接口定义语言为编写分布式对象定义制定了语法，Internet ORB 间协议则用于做出远程方法调用。另外，CORBA 还定义了一些可选的附带服务，包括一个命名服务、事件服务、时间服务和许多其他服务。

如今有许多 CORBA 标准的开源实现都可以免费使用。本章中的例子使用了 “omniORB” 框架，可以从 <http://omniORB.sourceforge.net/> 得到。

使用 CORBA 需要多个步骤，包括定义对象接口，“编译” 接口生成网络和串行化代码，定义类方法实现，编写一个服务器进程，并编写客户。这一节将以一个极其简单的分布式数据库为背景，详细分析上述步骤，在此客户可以访问一个数据库服务器，这个数据库服务器可能位于另一个进程，甚至在另一个节点上。CORBA 是一个功能强大但也相当复杂的体系结构，这里的讨论只是触及它的一点皮毛。如果有兴趣用它来建立分布式对象框架，可以参考附录 B 中所列的一些参考资料。

### 24.3.1 接口定义语言

CORBA 在将对象接口与其实现相分离这个方面做得很好。编写一个分布式 CORBA 类时，首先用接口定义语言定义接口。这个语言看上去和 C++ 很像，不过不完全相同。实际上，IDL 是独立于实现语言的。理论上讲，可以用 C++ 为类编写一个实现，再用 Java 编写一个使用这个类的客户。

#### 编写接口

在这一步中，要指定对象所实现方法的原型。不过，不同于 C++ 类定义，在此不用提供成员变量或其他实现细节。

例如，假设希望这个简单的分布式数据库存储键/值对记录，其中键和值都是 string。以下是一个支持两个方法的数据库的 IDL 文件：

```
// database.idl

interface database {
    void addRecord(in string key, in string record);
    string lookupRecord(in string key);
};
```

参数前面的“in”指定这是值参数而不是引用参数。

### 生成桩和骨架

编写对象接口后，用一个 IDL 编译器编译此 IDL 文件，这会生成远程过程调用和网络层。针对各种语言已经有许多可用的 IDL 编译器，包括 Java、Python、C 以及 C++（这是当然的）。这一步会生成两组文件：桩（stub）和骨架（skeleton）。

### 桩

前面介绍 PRC 时曾指出，桩就是客户端的对象方法，其中隐藏了向另一个机器做实际远程调用所需的网络和串行化代码。omniORB IDL 编译器将 IDL 文件 name.idl 的桩代码置于头文件 name.hh 和源文件 nameSK.cc 中。以下是 database.hh 中桩代码的一个小例子，这是由 database.idl 文件生成的：

```
// This file is generated by omniidl (C++ backend)- omniORB_4_0. Do not edit.

// <There's a lot more code than we show here.>

class _objref_database :
    public virtual CORBA::Object, public virtual omniObjRef
{
public:
    void addRecord(const char* key, const char* record);
    char* lookupRecord(const char* key);

    inline _objref_database() { _PR_setobj(0); } // nil
    _objref_database(omniIOR*, omniIdentity*);

protected:
    virtual ~_objref_database();

private:
    virtual void* _ptrToObjRef(const char*);

    _objref_database(const _objref_database&);
    _objref_database& operator = (const _objref_database&);
    // not implemented

    friend class database;
};
```

下面是 databaseSK.cc 中的一个方法实现：

```
// This file is generated by omniidl (C++ backend)- omniORB_4_0. Do not edit.

// <There's a lot more code than we show here.>
```

```

void _objref_database::addRecord(const char* key, const char* record)
{
    _ORL_cd_D115D31DB8E47435_00000000 _call_desc(_ORL_lcfm_D115D31DB8E47435_10000\
000, "addRecord", 10);
    _call_desc.arg_0 = key;
    _call_desc.arg_1 = record;

    _invoke(_call_desc);
}

```

现在不理解这段代码没关系，我们只是想提供一个例子，让你知道在“后台”会有哪些工作。

### 骨架

骨架是类实现的基础，通常是从 IDL 接口生成的抽象基类。omniORB 把骨架同样放在 database.hh 和 databaseSK.cc 文件中，桩代码也置于这两个文件中。以下是 database.hh 中的骨架代码：

```

class _impl_database :
    public virtual omniServant
{
public:
    virtual ~_impl_database();

    virtual void addRecord(const char* key, const char* record) = 0;
    virtual char* lookupRecord(const char* key) = 0;

public: // Really protected, workaround for x1C
    virtual _CORBA_Boolean _dispatch(omniCallHandle&);

private:
    virtual void* _ptrToInterface(const char*);
    virtual const char* _mostDerivedRepoId();
};

class POA_database :
    public virtual _impl_database,
    public virtual PortableServer::ServantBase
{
public:
    virtual ~POA_database();

    inline ::database_ptr _this() {
        return (::database_ptr) _do_this(::database::_PD_repoId);
    }
};

```

注意，IDL 文件中的一个 in string 参数在生成的 C++ 代码中翻译为一个 const char\*。POA 代表可移植对象适配器 (Portable Object Adapter)，这是管理服务端对象引用的一个 CORBA 组件。

### 24.3.2 实现类

既然已经定义了接口，并生成了桩和骨架，下一步是编写一个类，来提供 IDL 文件中方法的具体实现。可以派生抽象骨架类来编写这个类，并补入数据成员和方法实现。实现方法时不必操心串行化或网络代码。只需像正常方法一样编写就可以了。骨架代码会处理所有麻烦的 RPC 细节。以下是基于前面 omniORB 骨架代码的 DatabaseServer 类定义：

```
// DatabaseServer.h
#include "database.hh"
#include <map>
#include <string>

class DatabaseServer : public POA_database,
                      public PortableServer::RefCountServantBase
{
public:
    DatabaseServer();
    virtual ~DatabaseServer();
    virtual void addRecord(const char* key, const char* record);
    virtual char* lookupRecord(const char* key);

protected:
    std::map<std::string, std::string> mDb;
};
```

需要注意的是，这个类派生了前面的 POA\_database 骨架抽象类，还派生了框架所提供的一个引用计数混合类。它增加了一个 protected map 数据成员来存储键/值对。

以下是方法实现：

```
#include "DatabaseServer.h"
using namespace std;

DatabaseServer::DatabaseServer()
{
}

DatabaseServer::~DatabaseServer()
{
}

void DatabaseServer::addRecord(const char* key, const char* record)
{
    mDb[key] = record;
}

char* DatabaseServer::lookupRecord(const char* key)
{
    return (CORBA::string_dup(mDb[key].c_str()));
}
```

这些实现中惟一的难点是，要记住使用 CORBA::string\_dup() 方法来复制从 lookupRecord() 返回的 string。

### 24.3.3 使用对象

现在可以使用分布式对象了。使用对象要有两步：必须有一段代码创建一个对象，并通过可移植对象适配器向对象请求代理（Object Request Broker, ORB）框架注册。还必须提供一种方法让客户代码查找对象的引用。可以采用的一种技术是使用一个命名服务器（nameserver）。这个命名服务器必须对分布式程序运行所在的所有机器都可用。作为一个命名服务器，它会把名字映射至对象引用，并跟踪系统上所有分布式对象的实际物理位置。如果没有可用的命名服务器，可以使用其他特殊的方法来注册和查找

对象引用。尽管 CORBA 标准包括一个命名服务器，而且 omniORB 也提供了一个，不过为简单起见，我们的数据库例子只是把对象引用键写至一个文件。

使用对象的代码要在命名服务器中查找对象，或者（在我们这个例子中）在文件中查找，来获得对象的一个引用。客户代码对该引用调用一个方法时，请求会发送至 ORB 层，取决于具体实现，这可能是各进程中的一部分，也可能是各节点中自己的进程。此时有两个选择。如果底层对象与调用者在同一个进程中，此方法就会像一个正常的 C++ 方法调用一样在本地执行。不过，如果底层对象在同一个机器上的不同进程中，或者在一个远程机器上，ORB 则通过网络将方法请求发送到服务器进程。所有这些工作都在后台进行；对对象引用做方法调用的代码无需操心实际对象是一个本地对象还是远程对象。

图 24-3 显示了基本 CORBA 体系结构。

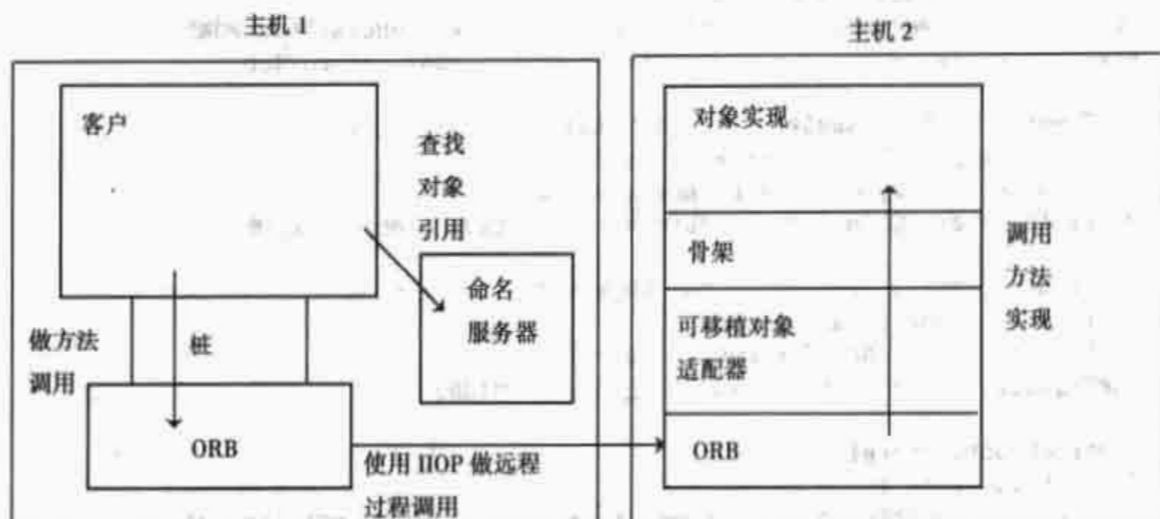


图 24-3

CORBA 可以用于同一机器上进程间的通信，也适用于机器间的通信。可以把 CORBA 用作一种在相同机器上不同进程之间“共享”对象的机制。

这一节余下的部分将继续完成这个数据库例子，我们会提供一个服务器和客户的示例实现。在此不指望你完全理解所有代码细节，只是希望提供一个例子，让你对 CORBA 编程有一个一般性的认识，并初步了解这项技术有多强大。如果想成为一个专业的 CORBA 程序员，可以参考附录 B 中所列的参考资料。

### 服务器进程

服务器进程必须初始化 ORB，创建一个新的 DatabaseServer 对象，注册此对象，并在文件中保存引用的一个键，以便客户查找。这个例子假设客户能够访问启动此服务器进程时所在的目录，可以通过一个网络文件系统来访问，也可能是因为它们本来就是在同一个节点上运行。这里的注释解释了所采取的各个步骤。需要注意，不必使用一个特殊的编译器来编写这段代码，可以使用一个标准的 C++ 编译器，如 g++，只要链接了适当的 omniORB 库即可。

```
#include "DatabaseServer.h"
#include <iostream>
#include <fstream>
using namespace std;

const char* objRefFile = "OBJ_REF_FILE.dat";
```

```

int main(int argc, char** argv)
{
    // Try to initialize the orb.
    CORBA::ORB_var orb;
    try {
        orb = CORBA::ORB_init(argc, argv);
    } catch(CORBA::SystemException&) {
        cerr << "Unable to initialize the ORB\n";
        exit(1);
    }

    // Obtain a reference to the "Portable Object Adapter" and downcast
    // it to the appropriate type.
    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

    // Create the DatabaseServer object and register/activate it
    // with the portable object adapter.
    DatabaseServer* myDb = new DatabaseServer();
    PortableServer::ObjectId_var dbid = poa->activate_object(myDb);

    // Write a string version of the object reference to a
    // file so clients can find us.
    CORBA::Object_var dbobj = myDb->_this();
    CORBA::String_var sior(orb->object_to_string(dbobj));

    ofstream ostr(objRefFile);
    if (ostr.fail()) {
        cerr << "Unable to open object reference file for writing.\n";
        exit(1);
    }
    ostr << (char*)sior;
    ostr.close();

    // Tell the reference counter that we're done with the object.
    // Now only the POA has a reference to it.
    myDb->_remove_ref();

    // Move the POA from holding to active state, so that it will process
    // incoming requests.
    PortableServer::POAManager_var pman = poa->the_POAManager();
    pman->activate();

    // Wait for incoming requests.
    orb->run();

    // Shouldn't return from the run call, but if we do, we need to clean up
    orb->destroy();
    return (0);
}

```

### 客户进程

最后一步是编写一个客户进程。以下是一个基本的客户，它从一个文件读取对象引用键，从这个键创建一个对象引用，并对该对象引用调用两个方法。这些调用会由 ORB 层翻译为对服务器进程中 DatabaseServer 对象的调用。



```
#include "database.hh"
#include <iostream>
#include <fstream>
using namespace std;

const char* objRefFile = "OBJ_REF_FILE.dat";

int main(int argc, char** argv)
{
    // Try to initialize the orb.
    CORBA::ORB_var orb;
    try {
        orb = CORBA::ORB_init(argc, argv);
    } catch(CORBA::SystemException&) {
        cerr << "Unable to initialize the ORB\n";
        exit(1);
    }

    // Read the server object reference from the file.
    ifstream istr(objRefFile);
    if (istr.fail()) {
        cerr << "No object reference file!\n";
        exit(1);
    }
    char objRef[1024];
    istr.getline(objRef, 1024);

    // Construct an object reference from the string.
    database_var dbref;
    try {
        CORBA::Object_var obj = orb->string_to_object(objRef);
        dbref = database::_narrow(obj);
        if(CORBA::is_nil(dbref) ) {
            cerr << "Can't narrow reference to type database\n";
            exit(1);
        }
    } catch(CORBA::SystemException&) {
        cerr << "Unable to find the object reference\n";
    }

    // Make calls on the object reference, which are translated to
    // calls on the server object in the server process.
    try {
        dbref->addRecord("key1", "value1");
        const char* lookup = dbref->lookupRecord("key1");
        if (strcmp(lookup, "value1") == 0) {
            cout << "Success!\n";
        } else {
            cout << "strings don't match\n";
        }
    } catch(CORBA::COMM_FAILURE&) {
        cerr << "Communication error\n";
        exit(1);
    } catch(CORBA::SystemException&) {
        cerr << "Communication error (SystemException)\n";
        exit(1);
    }
}
```

```
// We're done.  
orb->destroy();  
  
return (0);  
}
```

可以看到，CORBA 框架确实很复杂，学习 CORBA 的学习曲线很陡。不过，对于工业强度的分布式编程来说，CORBA 有着非凡的意义。

## 24.4 XML

可扩展标记语言（Extensible Markup Language, XML）是一个简单的通用标记语言。基本说来，XML 可以用于表示任何东西。例如，XML 可以作为文件格式来存储 MP3 音乐播放列表，或者作为一个复杂购买订单的内部表示。由于 XML 很容易使用，而且提供了跨平台支持，所以它迅速流行开来，成为网络通信、远程过程调用和分布式对象常采用的一种格式。

### 24.4.1 XML 快速入门

XML 最突出的一个特点就是它的学习曲线很平缓，这也是 XML 得到迅速采纳的原因之一。XML 入门很简单。只需几分钟，就能了解有关术语，并能读写合法的 XML。在此基础上，XML 开发人员可以继续深入，踏上一些更为复杂的道路，如 XML 转换，或简单对象访问协议（Simple Object Access Protocol, SOAP），这是建立在 XML 之上的一个分布式对象技术。

#### 什么是 XML

XML 只是一种表示数据的语法（syntax）。脱离特定的应用，XML 数据没有任何含义。例如，可以编写一个家庭财产管理程序，它能生成一个完全合法的 XML 文档来描述你的所有值钱的财物。如果把这个文档交给别人，他们自己可能能够查看并明确这个文档的内容，但是他们编写的基于 XML 的家庭财产管理程序则不一定能够解释这个文档。原因在于，XML 定义了文档的结构，但是没有定义其含义。另一个应用可能会用一种不同的结构表示同样的信息。

XML 以一种纯文本格式编写，这样就能很容易地理解。即使以前从未见过 XML，可能也能够理解以下这段 XML 数据：

```
<inventory>  
  <office>  
    <desk type="wood"/>  
    <computer type="Macintosh"/>  
    <chair type="leather"/>  
  </office>  
  <kitchen>  
    <mixer type="chrome"/>  
    <stove type="electric"/>  
  </kitchen>  
</inventory>
```

除了可读性外，文本格式还意味着，XML 可以很容易地用于软件中。不必学习一个复杂的框架或购买一个昂贵的工具包来解析文本。因为即使是最“差劲”的操作系统也能理解纯文本，所以可以很容易地在不同系统之间发送 XML，而无需操心二进制兼容问题。

平心而论，文本表示也存在一些缺点。可读性很快会导致冗长。数据的 XML 表示往往比相应的二进制表示大很多。数据集很大时，XML 表示可能会变得极其庞大。另外还要花费时间来解析文本，而二进

制格式根本无需任何解析。

根据前一个例子的缩进可以看出，XML 还有一个重要的特点，这就是它的层次性。可以看到，XML 通常解析为一个树结构，可以对这个树遍历来处理数据。

### XML 结构和术语

XML 文档的开头是一个文档序言 (document prolog)，它指定了文档的字符编码和其他元数据。许多程序员会忽略此文档序言，不过，一些比较严格的 XML 解析器如果在第一行没有看到文档序言，就不认为这个文档是一个 XML 文档。序言中可以指定哪些信息呢？有关的详细内容超出了本书的范围。以下序言对于大多数用途来讲都足够了：

```
<?xml version="1.0"?>
```

文档序言是一种特殊类型的标记 (tag)，标记是一种语法，XML 会认为它具有某种含义。如果你已经编写过 HTML 文件，应该对标记并不陌生。XML 文档的体就是由元素标记 (element tag) 组成的。它们只是一些记号，明确指定了一段逻辑结构的开始和结束。在 XML 中，每个开始元素标记都有一个与之对应的结束元素标记。例如，以下这行 XML 使用了标记 sentence 来标记一个句子元素的开始和结束。

```
<sentence>Let's go get some ice cream.</sentence>
```

在 XML 中，结束标记写作一个斜线后面跟有元素名。元素标记不一定像前例一样包含数据。在 XML 中，可以有一个空标记 (empty tag)，它只是独立存在 (译者注：即不包含数据)。为此可以在一个开始标记的后面紧接着一个结束标记：

```
<empty></empty>
```

XML 还为空元素标记提供了一个简写。如果标记的后面以一个斜线结束，它就同时用作此元素的开始标记和结束标记：

```
<empty />
```

最顶层的元素称为根元素 (root element)，它包含了文档中的所有其他元素。除了名字外，元素标记可以包含键/值对，这称为属性 (attribute)。哪些可以写作一个属性，对此没有固定的规则 (要记住，XML 只是一个语法而已)，不过一般地，属性提供了元素的元信息。例如，句子元素可以有一个属性来说明说这句话的人：

```
<sentence speaker="Marni">Let's go get some ice cream.</sentence>
```

元素可以有多个属性，不过它们都必须有惟一的键：

```
<sentence speaker="Marni" tone="pleading">Let's go get some ice cream.</sentence>
```

如果看到一个 XML 元素的名字中有一个冒号，如 <a:sentence>，冒号前面的字符串则是其命名空间 (namespace)。就像在 C++ 中一样，XML 中的命名空间允许分段使用名字。

在前面的例子中，元素的内容要么为空，要么为文本数据，这通常称为文本节点 (text node)。XML 中，元素也可以包含其他元素，这就使 XML 可以有层次结构。在以下例子中，对话元素由两个句子元素组成。要注意，这里的缩进只是为了便于阅读，XML 会忽略标记之间的空格。

```
<dialogue>
  <sentence speaker="Marni">Let's go get some ice cream.</sentence>
  <sentence speaker="Scott">After I'm done writing this C++ book.</sentence>
</dialogue>
```

基本知识就这么多。元素、属性和文本节点就是 XML 的构建模块。要了解更高级的语法，如特殊的字符转义，请参考附录 B 所列有关 XML 的参考书。

#### 24.4.2 XML 作为一种分布式对象技术

由于 XML 很简单，也很容易使用，所以已经作为一种串行化机制相当流行了。XML 串行化对象可以通过网络发送，发送者能够相信接收者肯定能解析这些对象，而不论接收者在什么平台上。例如，考虑以下所示的 Simple 类：

```
class Simple
{
public:
    std::string mName;
    int    mPriority;

    std::string mData;
};
```

类型为 Simple 对象可以串行化为以下 XML：

```
<Simple name="some name" priority="7">this is the data</Simple>
```

当然，由于 XML 没有指定应该如何使用各个节点，也可以如下简单地对其串行化：

```
<Simple name="some name" priority="7" data="this is the data" />
```

只要此串行化 XML 的接收者知道你采用什么规则来串行化此对象，接收者就能对其逆串行化。

与诸如 CORBA 等重量级的分布式对象技术相比，XML 串行化更为简单，知名度也越来越高。XML 的学习曲线比 CORBA 要缓和得多，同时也提供了同样的一些优点，如平台和语言独立性等。

#### 24.4.3 用 C++ 生成和解析 XML

由于 XML 只是一种文件格式，而不是一种对象描述语言，完成数据和 XML 转换的任务就要交给程序员了。一般地，编写 XML 非常容易。而且往往可以借助于一个第三方 XML 库来读取 XML。

##### 生成 XML

要将 XML 用作一种串行化技术，你的对象必须能够转换为 XML。在许多情况下，即时地构建一个 XML 流是输出 XML 最容易的方法。实际上，有一种提法，称 XML 元素“包装”在其他元素中，这个概念会让问题更简单。可以把新的 XML 文档建立为现有 XML 文档的合并文档。听上去有点复杂，请考虑以下例子。假设有一个名为 getNextSentenceXML() 的函数，它要求用户提供一个句子，并将其作为句子的一个 XML 表示返回。由于这个函数会返回句子作为一个合法的 XML 元素，就可以创建一个由句子组成的会话，即把多次 getNextSentenceXML() 调用的结果包装在一个会话元素标记中：

```
string getDialogueXML()
{
    stringstream outStream;
```



```

// Begin the dialogue element.
outStream << "<dialogue>";

while (true) {
    // Get the next sentence.
    string sentenceXML = getNextSentenceXML();
    if (sentenceXML == "") break;

    // Add the sentence element.
    outStream << sentenceXML;
}

// End the dialogue element.
outStream << "</dialogue>";

return outStream.toString();
}

```

如果后面的 `getNextSentenceXML()` 调用会返回前例中的句子，这个函数的结果就是：

```

<dialogue><sentence speaker="Marni">Let's go get some ice
cream.</sentence><sentence speaker="Scott">After I'm done writing this C++
book.</sentence></dialogue>

```

这个输出有点奇怪，因为这里没有用换行符和制表符调整格式。如果想对这个输出做一点美化，可以有几种选择：

- 可以使用一个第三方工具。例如，开源命令行程序 `tidy` (<http://tidy.sourceforge.net>) 的诸多有用的工具中就有一个 XML 美化打印 (pretty-print) 特性。
- 可以手工地在代码中加入回车和空格。这会很快变得复杂，因为在 `getNextSentenceXML()` 内部，代码不知道要使用多少个制表符。
- 可以使用 (或编写) 一个简单的 XML 生成类库，它了解嵌套的元素，并且会适当地调整格式。

#### 一个 XML 输出类

尽管输出 XML 很简单，但还是应该把 XML 输出代码重构到一个单独的类或一组类中，对此有许多充分的理由。除了前面所见的格式化问题外，分离出 XML 生成代码有以下几个好处：

- 更简洁的代码。谁想看到到处都是 `<` 呢?!
- 可以在这里集中实现特殊字符的转义。
- 这是一种更具面向对象性的方法。XML 可以是对象，对象就可以存储、传递至方法以及加以组织。
- 通过集中输出，可以减少可能的 XML 语法错误。

编写一个 XML 生成类也很简单。一个简单 XML Element 类的类定义如下所示：

```

// XMLElement.h

#include <string>
#include <vector>
#include <map>
#include <iostream>

class XMLElement
{

```

```

public:
    XMLElement();

    void setElementName(const std::string& inName);

    void setAttribute(const std::string& inAttributeName,
                     const std::string& inAttributeValue);

    void addSubElement(const XMLElement* inElement);

    // Setting a text node will override any nested elements.
    void setTextNode(const std::string& inValue);

    friend std::ostream& operator<<(std::ostream& outStream,
                                   const XMLElement& inElem);

protected:
    void writeToStream(std::ostream& outStream, int inIndentLevel = 0) const;

    void indentStream(std::ostream& outStream, int inIndentLevel) const;

private:
    std::string                mElementName;
    std::map<std::string, std::string> mAttributes;
    std::vector<const XMLElement*> mSubElements;
    std::string                mTextNode;
};

```

使用这个类，用户可以很容易地创建 XMLElement 对象、设置其属性并设置文本节点或子元素。无论什么时候，客户都可以调用 operator<< 来得到元素当前状态的 XML 表示。

下面提供一个示例实现。由于它使用了 C++ 语法，你现在应该对此很精通了，所以我们不再逐行地进行解释。如果刚开始有些不明白，可以参考在此提供的内联注释。

```

#include "XMLElement.h"

using namespace std;

XMLElement::XMLElement() : mElementName("unnamed")
{
}

void XMLElement::setElementName(const string& inName)
{
    mElementName = inName;
}

void XMLElement::setAttribute(const string& inAttributeName,
                              const string& inAttributeValue)
{
    // Set the key/value pair, replacing the existing one if it exists.
    mAttributes[inAttributeName] = inAttributeValue;
}

void XMLElement::addSubElement(const XMLElement* inElement)
{
}

```



```

    // Add the new element to the vector of subelements.
    mSubElements.push_back(inElement);
}

void XMLElement::setTextNode(const string& inValue)
{
    mTextNode = inValue;
}

ostream& operator<<(ostream& outStream, const XMLElement& inElem)
{
    inElem.writeToStream(outStream);
    return (outStream);
}

void XMLElement::writeToStream(ostream& outStream, int inIndentLevel) const
{
    indentStream(outStream, inIndentLevel);
    outStream << "<" << mElementName; // open the start tag

    // Output any attributes.
    for (map<string, string>::const_iterator it = mAttributes.begin();
         it != mAttributes.end(); ++it) {
        outStream << " " << it->first << "=\"" << it->second << "\"";
    }

    // Close the start tag.
    outStream << ">";

    if (mTextNode != "") {
        // If there's a text node, output it.
        outStream << mTextNode;
    } else {
        outStream << endl;
        // Call writeToStream at inIndentLevel+1 for any subelements.
        for (vector<const XMLElement*>::const_iterator it = mSubElements.begin();
             it != mSubElements.end(); ++it) {
            (*it)->writeToStream(outStream, inIndentLevel + 1);
        }
        indentStream(outStream, inIndentLevel);
    }

    // Write the close tag.
    outStream << "</" << mElementName << ">" << endl;
}

void XMLElement::indentStream(ostream& outStream, int inIndentLevel) const
{
    for (int i = 0; i < inIndentLevel; i++) {
        outStream << "\t";
    }
}

```

前面的实现可以作为一个不错的起点，而且对于简单的 XML 应用来说也很理想。这里还少几个特性，其中之一就是没有完成特殊字符的转义。例如，字符 & 需要在 XML 文档中转义为 & amp;。以下示

例程序显示了如何使用 XMLElement 类来建立前例手工输出的文档：

```
int main(int argc, char** argv)
{
    XMLElement dialogueElement;
    dialogueElement.setElementName("dialogue");

    XMLElement sentenceElement1;
    sentenceElement1.setElementName("sentence");
    sentenceElement1.setAttribute("speaker", "Marni");
    sentenceElement1.setTextNode("Let's go get some ice cream.");

    XMLElement sentenceElement2;
    sentenceElement2.setElementName("sentence");
    sentenceElement2.setAttribute("speaker", "Scott");
    sentenceElement2.setTextNode("After I'm done writing this C++ book.");

    // Add the sentence elements as subelements of the dialogue element.
    dialogueElement.addSubElement(&sentenceElement1);
    dialogueElement.addSubElement(&sentenceElement2);

    // Output the dialogue element to stdout.
    cout << dialogueElement;

    return 0;
}
```

这个程序的输出如下：

```
<dialogue>
  <sentence speaker="Marni">Let's go get some ice cream.</sentence>
  <sentence speaker="Scott">After I'm done writing this C++ book.</sentence>
</dialogue>
```

许多 XML 解析库还包括 XML 输出工具。如果使用一个 XML 解析器完成输入（后面将说明），在自行编写输出功能之前先看看它是不是已经提供了输出功能。

### 解析 XML

要对 XML 对象逆串行化，需要解释或解析（parse）文档。除非你读到的 XML 极为简单而且得到了严格定义，否则很可能想使用一个第三方 XML 解析库。XML 解析库通常有两种形式：SAX 和 DOM。

SAX（XML 的简单 API，Simple API for XML）解析器使用一个基于事件的解析模型。要使用一个 SAX 解析器，需要注册回调函数或者一个实现了某些方法的对象。解析文档时，就会调用适当的函数或方法，使你有机会完成某个动作。例如，如果想查找一个文档中重复出现的 XML 元素名，可以注册一个回调，到达一个元素开始标记时就会触发这个回调。在内部，可以维护一个已遇到元素的列表。使用这个列表，就可以检测出重复情况。

DOM（文档对象模型，Document Object Model）解析器将一个 XML 文档转换为一个树状结构，可以很容易地通过代码对其遍历。如果程序员熟悉面向对象层次体系和树数据结构，DOM 方法则更为自然。DOM 方法的缺点是效率不高。由于它会解析整个文档，并建立一个结构，所以一般会比 SAX 要慢，而且更耗费内存。尽管这一节后面只处理 DOM 解析器，但是你会发现大多数 XML 解析器都同时支持 SAX 和 DOM。

### Xerces XML 库

最流行的 XML 解析器之一是 Xerces，这是 Apache XML 项目的一部分。Xerces 是一个开源解析器，对许多语言都可用，其中也包括 C++。可以从 <http://xml.apache.org/> 下载 Xerces-C++ 库。

安装了 Xerces 并增加到 C++ 工程后，你就不用负责解析 XML 的工作了。尽管 Xerces 提供了大量的功能，但仍然很容易着手使用，由此说明这是一个设计得很好的库！

Xerces DOM 解析器中最重要的类是 DOMNode。DOMNode 是 XML 数据的一个单元，可能包括其他节点。DOMNode 类的子类包括 DOMDocument、DOMElement、DOMAttr、DOMText 等等。使用 Xerces DOM 通常要从根节点（一个 DOMDocument）开始，遍历节点树来找到所需的数据。图 24-4 显示了 <dialogue> XML 文档节点树的一个稍加简化的版本。这里的简化是指它只显示了确实包含数据的节点。

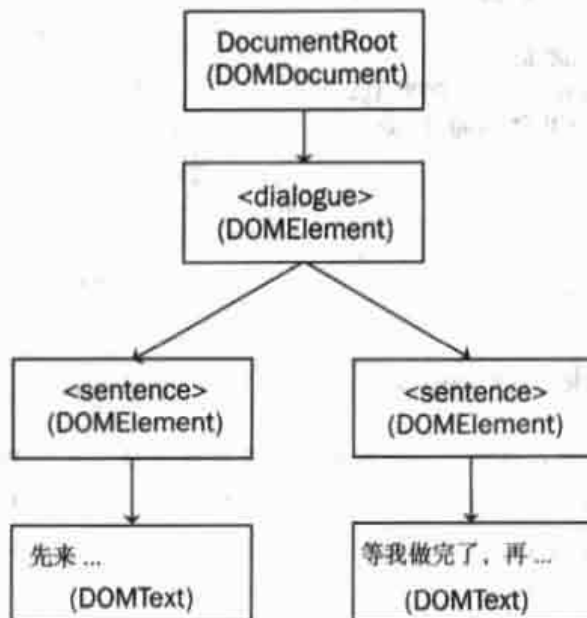


图 24-4

图 24-4 中没有显示 XML 属性，因为属性就是元素的特性，而不是元素的子元素。

### 使用 Xerces

首先要面对的一个难点是，Xerces 如何表示字符串。由于 XML 可以采用多种方式编码，库有其自己的字符类型：XMLch。它还有一个工具类，名为 XMLString，利用这个工具类可以更容易地使用 XMLch 字符串，以及将其转换为我们更为熟悉的 char。例如，如果一个 Xerces 方法把数据作为一个 XMLch\* 字符串返回，就可以使用 XMLString::transcode() 将其输出，得到一个 C 风格的字符串：

```
void outputXercesString(XMLch* inXercesString)
{
    char* familiarString = XMLString::transcode(inXercesString);
    cout << familiarString << endl;
}
```

由于 transcode() 会为 C 风格的字符串分配内存，所以还必须用 XMLString::release() 释放所分配的空间，这个方法取一个 C 风格字符串的指针作为参数（这个设计有点奇怪）。以下修改后的版本可以避免内存泄漏：

```
void outputXercesString(XMLch* inXercesString)
{
```

```
char* familiarString = XMLString::transcode(inXercesString);
cout << familiarString << endl;
XMLString::release(&familiarString);
}
```

了解了上述特殊情况后，下面该解析一些 XML 了。这个例子将一个名为 test.xml 的文件解析到一个 DOM 树中，然后循环处理所有节点，打印出所遇到的所有元素的名、这些元素中包含的所有属性以及所有文本节点的内容。

这个程序的开头包括必要的标准首部和 Xerces 首部。它还声明了 XERCES\_CPP\_NAMESPACE\_USE，这是 Xerces 中的一个 #define，给出了此文件正确的命名空间。

```
#include <xercesc/util/PlatformUtils.hpp>

#include <xercesc/dom/DOM.hpp>
#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/util/XMLString.hpp>

#include <iostream>

XERCES_CPP_NAMESPACE_USE
using namespace std;

void printNode(const DOMNode* inNode);
```

这个程序的 main() 相当简单，尽管实际的解析正是在这里完成。它先初始化 Xerces 库。接下来，创建一个新的 DOM 解析器，并告诉它解析此文件。这个操作的结果是一个表示此完整文档的 DOMNode。为了得到根元素，要调用 getDocumentElement()。这个值传递至 printNode()，此函数遍历整个树，打印出相应数据。最后，程序会在退出之前清除 XML 库。

```
int main(int argc, char** argv)
{
    XMLPlatformUtils::Initialize();

    XercesDOMParser* parser = new XercesDOMParser();
    parser->parse("test.xml");

    DOMNode* node = parser->getDocument();
    DOMDocument* document = dynamic_cast<DOMDocument*>(node);
    if (document != NULL) {
        printNode(document->getDocumentElement());
    }

    delete parser;
    XMLPlatformUtils::Terminate();

    return 0;
}
```

printNode() 函数最有意思。由于参数 inNode 可以是任意类型的 XML 节点，此函数会按顺序尝试两种已知的节点类型。首先，它会尝试动态地将节点类型强制转换为一个文本节点，如果节点是另一种类型，则捕获强制类型转换错误。

```

void printNode(const DOMNode* inNode)
{
    try {
        const DOMText& textNode = dynamic_cast<const DOMText&>(*inNode);
        char* text = XMLString::transcode(textNode.getData());
        cout << "Found text data: " << text << endl;
        XMLString::release(&text);
    } catch (bad_cast) {
        // Not a text node . . .
    }
}

```

接下来，它再尝试将节点强制类型转换为一个元素节点。如果这个强制类型转换成功，元素的名和所包含的所有属性都会打印出来。

```

    try {
        const DOMELEMENT& elementNode = dynamic_cast<const DOMELEMENT&>(*inNode);
        char* tagName = XMLString::transcode(elementNode.getTagName());
        cout << "Found tag named: " << tagName << endl;
        XMLString::release(&tagName);

        // Look at the attribute list.
        DOMNamedNodeMap* attributes = elementNode.getAttributes();
        for (int i = 0; i < attributes->getLength(); i++) {
            try {
                const DOMAttr& attrNode =
                    dynamic_cast<const DOMAttr&>(*attributes->item(i));
                char* name = XMLString::transcode(attrNode.getName());
                char* value = XMLString::transcode(attrNode.getValue());
                cout << "Found attribute pair: (" << name << "=" << value << ")"
                    << endl;
                XMLString::release(&name);
                XMLString::release(&value);
            } catch (bad_cast) {
                cerr << "Error converting attribute!" << endl;
            }
        }
    } catch (bad_cast) {
        // Not an element node . . .
    }
}

```

最后，此函数会对子节点递归地调用 printNode()。在实际中，只在元素节点上才存在子节点。

```

// Print any subelements.
DOMNodeList* children = inNode->getChildNodes();
for (int i = 0; i < children->getLength(); i++) {
    printNode(children->item(i));
}

```

如果提供<dialogue>文档作为输入，这个程序将生成以下输出：

```

Found tag named: dialogue
Found text data:

```



```

Found tag named: sentence
Found attribute pair: (speaker=Marni)
Found text data: Let's go get some ice cream.
Found text data:

Found tag named: sentence
Found attribute pair: (speaker=Scott)
Found text data: After I'm done writing this C++ book.
Found text data:

```

要注意标记之间的空格读作为一个文本节点。

Xerces 库存在异常。前面的例子遇到一个异常而不是 `bad_cast` 时会中止，不过成品质量的应用应该捕获并处理 Xerces 异常。

#### 24.4.4 XML 验证

XML 是一种通用语法，没有预定义的标记，也没有自己的语义。不过，这并不表示任何 XML 应用都能解释任何 XML 输入。编写一个处理 XML 的应用时，需要指定所能解释的特定类型的 XML。XML 验证允许定义应用所允许的 XML 的特定格式，包括元素名、元素组织及元素的属性。

##### DTD (文档类型定义)

文档类型定义 (Document Type Definition) 是指定 XML 文档类型的原始技术。DTD 的结构看上去很像 XML 本身的结构，但并非如此。DTD 不采用层次格式，而是编写为有关文档类型的一系列声明。有关 DTD 创建的详细内容超出了本书的范围。不过，为了让你对 DTD 是什么样子有一个认识，以下给出 `<dialogue>` 文档的相应 DTD：

```

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT dialogue (sentence+)>
<!ELEMENT sentence (#PCDATA)>
<!-- ATTLIST sentence
speaker (Marni | Scott) #REQUIRED
-->

```

在一个 XML 文档内部，可以在文件最上面包括一个 DOCTYPE 断言，指定该文件遵循哪一个 DTD：

```

<?xml version="1.0"?>
<!DOCTYPE dialogue SYSTEM "dialogue.dtd">
<dialogue>
  <sentence speaker="Marni">Let's go get some ice cream.</sentence>
  <sentence speaker="Scott">After I'm done writing this C++ book.</sentence>
</dialogue>

```

DOCTYPE 断言需要两个参数。第一个是文档的根元素，第二个是 DTD 文件的位置。在这个例子中，DTD 位于本地系统上的一个文件 `dialogue.dtd` 中。

大多数 XML 解析库（包括 Xerces）都可以针对 XML 文件的 DTD 进行验证。这样一来，就可以保证程序只对它能解释的数据进行操作。

##### XML 模式

XML 文档的验证是一个不错的想法，但是 DTD 格式还有许多弱点。对于复杂的文档，DTD 会很快变得非常庞大。而且 DTD 没有提供相关工具来定义复杂类型、排序或数据内容。还有一点，DTD 甚至



不是用 XML 编写的。

XML 模式 (XML Schema) 力图提供一种功能更强的方法来定义 XML 文档的类型。XML 模式定义比 DTD 灵活得多, 不过增加的灵活性也带来了更多的复杂性。有关 XML 模式有许多很棒的书 (见附录 B), 所以这里也只是提供一个非常简单的例子。

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="dialogue">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="sentence" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="sentence">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="speaker" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:NMTOKEN">
                <xs:enumeration value="Marni"/>
                <xs:enumeration value="Scott"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

就像 DTD 一样, 也可以将 XML 模式与一个 XML 文档相关联。但不是使用一个 DOCTYPE 声明, 而是在根元素的一个属性中指定模式的位置:

```
<?xml version="1.0"?>
<dialogue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="dialogue.xsd">
  <sentence speaker="Marni">Let's go get some ice cream.</sentence>
  <sentence speaker="Scott">After I'm done writing this C++ book.</sentence>
</dialogue>
```

需要注意, xmlns:xsi 属性指定这个文档是一个 XML 模式的实例, 该 XML 模式位于 xsi:noNamespaceSchemaLocation 属性指定的位置上。

利用一些软件包, 如 Altova Software ([www.xmlspy.com](http://www.xmlspy.com)) 的 xmlspy, 可以大大简化生成和解释 XML、XML 模式和 DTD 的过程。

#### 24.4.5 用 XML 构建分布式对象

XML 分布式对象就是一个对象, 它知道如何作为 XML 输出, 以及如何根据 XML 来填充。在这一节中, 我们将使用 XML 串行化把前面定义的 Simple 类转变成为一个分布式对象。

## XMLSerializable 混合类

在一个处理多个分布式对象的应用中，如果这些对象有一个共同的父类往往会很方便。下例中定义的 XMLSerializable 类要求其子类实现必要的方法，从而可以从 XML 读取对象，以及将对象写入 XML。这是一个混合类的例子（更多详细内容请见第 25 章）。

```
class XMLSerializable
{
public:
    virtual std::string toXML() = 0;
    virtual void fromXML(const std::string& inXML) = 0;
};
```

下面是新 Simple 类，继承自 XMLSerializable：

```
class Simple : public XMLSerializable
{
public:
    std::string mName;
    int mPriority;

    std::string mData;

    virtual std::string toXML();
    virtual void fromXML(const std::string& inXML);
};
```

## 实现 XML 串行化

利用前面实现的 XMLElement 类，并使用 Xerces 库，这对编写具体的串行化代码大有帮助：

```
string Simple::toXML()
{
    XMLElement simpleElement;
    simpleElement.setElementName("simple");

    simpleElement.setAttribute("name", mName);

    // Convert the int into a string.
    ostringstream tempStream;
    tempStream << mPriority;
    simpleElement.setAttribute("priority", tempStream.str());

    // Add the data as a text node.
    simpleElement.setTextNode(mData);

    // Convert the XMLElement into a string.
    ostringstream resultStream;
    resultStream << simpleElement;

    return resultStream.str();
}
```

```
void Simple::fromXML(const string& inString)
{
    static const char* bufID = "simple buffer";

    // Use MemBufInputSource to read the XML content from a string.
    MemBufInputSource src((const XMLByte*)inString.c_str(),
```

```

        inString.length(), bufID);
XercesDOMParser* parser = new XercesDOMParser();
parser->parse(src);

DOMNode* node = parser->getDocument();
DOMDocument* document = dynamic_cast<DOMDocument*>(node);
if (document == NULL) {
    delete parser;
    return;
}

// Document should be the <simple> element.
try {
    const DOMELEMENT& elementNode =
        dynamic_cast<const DOMELEMENT&>(*document->getDocumentElement());
    // Get the name attribute.
    XMLCh* nameKey = XMLString::transcode("name");
    char* name = XMLString::transcode(elementNode.getAttribute(nameKey));
    XMLString::release(&nameKey);
    mName = name;
    XMLString::release(&name);

    // Get the priority attribute.
    XMLCh* priorityKey = XMLString::transcode("priority");
    char* priorityStr =
        XMLString::transcode(elementNode.getAttribute(priorityKey));
    XMLString::release(&priorityKey);
    // Parse the priority number.

    istringstream tmpStream(priorityStr);
    tmpStream >> mPriority;

    XMLString::release(&priorityStr);

    // Get the data as a text node.
    const XMLCh* textData = elementNode.getTextContent();
    char* data = XMLString::transcode(textData);
    mData = data;
    XMLString::release(&data);
} catch (bad_cast) {
    cerr << "cast exception while parsing Simple object from XML" << endl;
} catch (...) {
    cerr << "an unknown error occurred while parsing a Simple object from XML"
        << endl;
}

delete parser;
}

```

以下是一个 main()，其中会创建一个 Simple 对象，将其写为 XML，然后把同样的 XML 输出读入一个新的 Simple 对象，由此来测试串行化功能。完成时，这两个对象应该相同。

```

int main(int argc, char** argv)
{
    XMLPlatformUtils::Initialize();

```

```
Simple test;
test.mName = "myname";
test.mPriority = 7;
test.mData = "my data";

string xmlData = test.toXML();

Simple test2;
test2.fromXML(xmlData);
if (test.mName == test2.mName) {
    cout << "Names are equivalent!" << endl;
} else {
    cout << "ERROR: Names are not equivalent!" << endl;
}

if (test.mPriority == test2.mPriority) {
    cout << "Priorities are equivalent!" << endl;
} else {
    cout << "ERROR: Priorities are not equivalent!" << endl;
}

if (test.mData == test2.mData) {
    cout << "Data is equivalent!" << endl;
} else {
    cout << "ERROR: Data is not equivalent!" << endl;
}

XMLPlatformUtils::Terminate();

return 0;
}
```

### 使用分布式对象

既然 Simple 对象可以从 XML 读取, 而且可以写为 XML, 这些对象就是完全 XML 可串行化的。XML 串行化是把 XML 用作分布式对象技术的基础。另一个难点是在不同机器和应用之间传输 XML 串行化对象。

与本章前面介绍的传统串行化机制一样, 可以针对任何网络或数据交换技术使用 XML 串行化。可以编写一个程序通过电子邮件传送串行化对象, 或者将其压缩, 作为二进制数据在网络上发送。由于 XML 只是一种语法, 就要由程序员来确定 XML 内容的实际语义及其传输机制。

### 24.4.6 SOAP (简单对象访问协议)

XML 最流行的一个应用就是通过网络交换数据。你已经了解了, XML 特别适合这种应用, 因为 XML 很易于使用, 而且所有平台都能接受。主要缺点是, 通过 XML 转换数据的应用必须对所交换 XML 数据的特定语义达成共识。如果只能用 XML, 在没有得到 XML 格式的情况下, 就无法编写应用来对别人的应用做 RPC 型调用。

SOAP 是一个用于交换数据的基于 XML 的标准。它提供了一种标准方法来做出 RPC 型请求、提供有关 XML 的元数据、用 XML 表示简单和复杂的数据类型 (使用 XML 模式) 以及处理错误。通过使用基于 SOAP 的 XML 作为数据交换格式, 应用可以轻松地通信, 而不用自行实现所有这些细节。

## SOAP 简介

本节将介绍 SOAP 中所用的一些术语，但并不深入分析语法的细节。要了解如何实现 SOAP 应用的有关详细内容，最好参考专门介绍 SOAP 的参考书。另外，大量新兴的 SOAP 框架和硬件设施的出现，使得程序员不必再考虑 SOAP 语法的细节，这些 SOAP 框架和硬件设施已经把 SOAP 语法包装在程序化或图形界面中。因此，尽管可能必须查看原始 SOAP 数据来完成调试，但是一般不太需要自己亲自编写。

SOAP 消息中的所有数据都包含在一个 SOAP 信封 (SOAP Envelope) 中。这个信封分为两部分：SOAP 首部 (SOAP Header) 和 SOAP 体 (SOAP Body)。可以想见，SOAP 首部 (Header) 包含了消息的元信息。例如，由于 XML 是一种纯文本的可读格式，在通过网络传输时极有可能遭到恶意修改。首部可以包含数字签名，用于验证一个 SOAP 消息的完整性。

SOAP 体 (Body) 的内容根据所用 SOAP 的样式不同会有所不同。文档型的 SOAP (Document-style SOAP) 消息只是在 SOAP 体中提供一个 XML 负载。如果应用希望使用 SOAP 标准将 XML 串行化数据从一个机器移至另一个机器，这些应用大多会利用文档型的 SOAP。以下是一个文档型 SOAP 消息的例子：

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <dialogue>
      <sentence speaker="Marni">Let's go get some ice cream.</sentence>
      <sentence speaker="Scott">After I'm done writing this C++
book.</sentence>
    </dialogue>
  </soap:Body>
</soap:Envelope>
```

PRC 型 SOAP (RPC-style SOAP) 是一种更结构化的 SOAP 消息，可用于向远程机器发出请求，并接收远程机器的响应。在一个 RPC 型请求中，SOAP 体包含了对远程机器所做请求的一个描述，包括请求的参数。以下是一个简单的 RPC 请求，要求调用一个完成两个数字相加的方法：

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <myNS:AddNumbers xmlns:myNS="mynamespace">
      <myNS:arg1>7</myNS:arg1>
      <myNS:arg2>4</myNS:arg2>
    </myNS:AddNumbers>
  </soap:Body>
</soap:Envelope>
```

对 RPC 型请求的响应中，SOAP Body 包含一个 XML 元素，其中包含了这个 RPC 调用的结果：

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <myNS:AddNumbersResponse xmlns:myNS="mynamespace">
      <myNS:result>11</myNS:result>
    </myNS:AddNumbersResponse>
  </soap:Body>
</soap:Envelope>
```

尽管有些语法有点不好懂，但你应该对 SOAP 消息中会包含些什么有一个很好的认识了。也应该了解到使用 SOAP 实现分布式应用的强大能力，只需使用简单的 XML，就可以从任何采用 SOAP 的应用发出请求和接收响应。这里不需要精巧、昂贵或特定于平台的技术。与非 SOAP 串行化 XML 相比，SOAP

还有一个优点，因为其语义是指定的。如果两个人想编写通过串行化 XML 通信的应用，他们就必须对属性和元素名达成一致意见。如果使用 SOAP，则可以把重点放在应用的具体内容上。

SOAP 作为企业和 Web 服务的一种数据交换机制已经日益普及。许多现有的 SOAP 框架都是为 Java 编写的（但 Microsoft 的 .NET 是一个突出的例外，它是以 C# 为目标）。不过，SOAP 并不特定于平台。实际上，SOAP 可以作为一种绝佳的方法将 C++ 应用提供给使用其他语言的更多用户。

## 24.5 小结

在本章中，你已经看到了如何使用网络技术编写新型的应用。在此了解了分布式通信的机制：串行化和远程过程调用。另外我们还介绍了实现这些技术的多种方法，包括定制串行化、CORBA、XML 和 SOAP。

分布式计算为应用开创了一个新的世界。既然已经知道了这些概念和各种技术，你就具备了编写分布式应用的前提条件，没准下一个流行的分布式应用就会是你编写的呢！



## 第 25 章 结合技术和框架

这本书的一大主题就是采用可重用技术和模式。作为一个程序员，你很可能会重复地遇到类似的问题。如果能配备一个“工具库”，准备各种各样不同的方法，通过对给定问题应用适当的技术，就能节省时间。

本章将关注设计技术，在此会介绍一些 C++ 惯用方法，它们并不是 C++ 语言必要的内置部分，但是使用相当频繁。这一章的第一部分会谈谈到 C++ 的一些常用的语言特性，不过其语法很容易被忘记。这里的大部分内容都只是一个复习，不过如果忘记了相应语法，可以把这当作一个有用的参考工具。这一部分涉及的主题包括：

- 从头开始建立一个类
- 用子类扩展一个类
- 抛出和捕获异常
- 读文件
- 写文件
- 定义模板类

本章的第二部分强调了在 C++ 语言特性之上建立的一些更高级的技术。这些技术可以提供一种更好的办法来完成每天的编程任务。这一部分介绍的内容包括：

- 带引用计数的智能指针
- 双重分派技术
- 混合类

这些概念在前面的许多章节中已经提到过，但是没有提供详细的代码例子。本章将针对这些概念提供具体的例子，有关的代码还可以用于自己的程序中。

本章的最后将介绍框架，这种编码技术可以大大简化大型应用的开发。

### 25.1 “我想不起来如何…”

第 1 章曾对 C 标准和 C++ 标准的大小做过比较。C 程序员有可能会记住 C 语言的全部方方面面，而且这种情况并不少见。C 语言中，关键字不多，语言特性很少，而且行为都得到了明确定义。而 C++ 就不是这么一回事了。即使是本书的作者（自认为是天才）也需要时不时地查阅有关内容。考虑到这一点，我们提供了以下编码技术的例子，这些技术几乎在所有 C++ 程序中都用到过。如果你能想到某个概念，但是忘记了它的语法，可以翻看这些例子来恢复记忆。

#### 25.1.1 ……编写一个类

不记得怎么开始了吗？没关系，以下是一个简单类的定义：

```

/**
 * Simple.h
 *
 * A simple class that illustrates class definition syntax.
 *
 */
#ifndef _simple_h_
#define _simple_h_

class Simple {

public:
    Simple();           // Constructor
    virtual ~Simple();  // Destructor

    virtual void publicMethod(); // Public method

    int mPublicInteger;    // Public data member

protected:
    int mProtectedInteger; // Protected data member

private:
    int mPrivateInteger;    // Private data member

    static const int mConstant = 2; // Private constant

    static int sStaticInt;         // Private static data member

    // Disallow assignment and pass-by-value
    Simple(const Simple& src);
    Simple& operator=(const Simple& rhs);
};
#endif

```

接下来，以下是类实现，这里包括静态数据成员的初始化：

```

/**
 * Simple.cpp
 *
 * Implementation of a simple class
 *
 */
#include "Simple.h"

int Simple::sStaticInt = 0; // Initialize static data member.

Simple::Simple()
{
    // Implementation of constructor
}

Simple::~Simple()
{
    // Implementation of destructor
}

```

```
void Simple::publicMethod()  
{  
    // Implementation of public method  
}
```

### 25.1.2 .....派生一个现有类

要派生一个类，需要声明一个新类，这是其父类的一个公共扩展。以下是一个名为 SubSimple 的示例子类的定义：

```
/**  
 * SubSimple.h  
 *  
 * A subclass of the Simple class  
 *  
 */  
  
#ifndef _subsimple_h_  
#define _subsimple_h_  
  
#include "Simple.h"  
  
class SubSimple : public Simple  
{  
public:  
    SubSimple();           // Constructor  
    virtual ~SubSimple();  // Destructor  
  
    virtual void publicMethod(); // Overridden method  
  
    virtual void anotherMethod(); // Added method  
};  
  
#endif
```

实现如下：

```
/**  
 * SubSimple.cpp  
 *  
 * Implementation of a simple subclass  
 *  
 */  
  
#include "SubSimple.h"  
  
SubSimple::SubSimple() : Simple()  
{  
    // Implementation of constructor  
}  
  
SubSimple::~~SubSimple()  
{  
    // Implementation of destructor  
}
```

```
void SubSimple::publicMethod()
{
    // Implementation of overridden method
}

void SubSimple::anotherMethod()
{
    // Implementation of added method
}
```

### 25.1.3 ……抛出和捕获异常

如果你参与一个团队开发，而且团队中没有使用异常，或者如果你已经习惯了 Java 风格的异常，很可能就会忘记 C++ 的异常语法。以下是一个简单的复习，在此使用了内置的异常类 `std::runtime_error`。在大多数大型程序中，要编写自己的异常类：

```
#include <stdexcept>
#include <iostream>
void throwIf(bool inShouldThrow) throw (std::runtime_error)
{
    if (inShouldThrow) {
        throw std::runtime_error("Here's my exception");
    }
}

int main(int argc, char** argv)
{
    try {
        throwIf(false); // doesn't throw
        throwIf(true);  // throws!
    } catch (const std::runtime_error& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }
}
```

### 25.1.4 ……读文件

有关文件输入的完整细节请见第 14 章。以下是展示读文件基础知识的一个简单示例程序。这个程序读取自己的源代码，而且一次输出一个 token（词法单位）。

```
/**
 * readfile.cpp
 */

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    ifstream inFile("readfile.cpp");
```

```
if (inFile.fail()) {
    cerr << "Unable to open file for reading." << endl;
    exit(1);
}

string nextToken;
while (inFile >> nextToken) {
    cout << "Token: " << nextToken << endl;
}

inFile.close();

return 0;
}
```

### 25.1.5 .....写文件

下面这个程序把一个消息写到一个文件，然后重新打开文件，并追加另一个消息。其他详细内容请见第 14 章。

```
/**
 * writefile.cpp
 */
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ofstream outFile("writefile.out");

    if (outFile.fail()) {
        cerr << "Unable to open file for writing." << endl;
        exit(1);
    }

    outFile << "Hello!" << endl;

    outFile.close();

    ofstream appendFile("writefile.out", ios_base::app);

    if (appendFile.fail()) {
        cerr << "Unable to open file for writing." << endl;
        exit(1);
    }

    appendFile << "Append!" << endl;

    appendFile.close();
}
```

### 25.1.6 .....编写模板类

模板语法是 C++ 语言中最麻烦的部分之一。有关模板最容易忘记的是：使用模板的代码不仅要能够

看到类模板定义，还要能够看到方法实现。为此，一般采用的技术是在头文件中用 `#include` 包含源文件，这样客户只需正常地用 `#include` 包含头文件就可以了。以下程序显示了一个类模板，它只是包装了一个对象，并为之增加了获取 (get) 和设置 (set) 语义。

```
/**
 * SimpleTemplate.h
 */

template <typename T>
class SimpleTemplate
{
public:
    SimpleTemplate(T& inObject);

    const T& get();
    void set(T& inObject);

protected:
    T& mObject;
};

#include "SimpleTemplate.cpp" // Include the implementation!
```

```
/**
 * SimpleTemplate.cpp
 */

template<typename T>
SimpleTemplate<T>::SimpleTemplate(T& inObject) : mObject(inObject)
{
}

template<typename T>
const T& SimpleTemplate<T>::get()
{
    return mObject;
}

template<typename T>
void SimpleTemplate<T>::set(T& inObject)
{
    mObject = inObject;
}
```

```
/**
 * TemplateTest.cpp
 */

#include <iostream>
#include <string>

#include "SimpleTemplate.h"

using namespace std;
```



```
int main(int argc, char** argv)
{
    // Try wrapping an integer.
    int i = 7;
    SimpleTemplate<int> intWrapper(i);
    i = 2;
    cout << "wrapper value is " << intWrapper.get() << endl;

    // Try wrapping a string.
    string str = "test";
    SimpleTemplate<string> stringWrapper(str);
    str += "!";
    cout << "wrapper value is " << stringWrapper.get() << endl;
}
```

## 25.2 还有更好的办法

此时此刻，全世界成千上万的程序员可能正在着力解决一些早已经得到解决的问题。圣约瑟的某个人可能在从头编写一个使用引用计数的智能指针实现。地中海群岛上的一个年轻程序员可能正在设计一个类层次体系，要是能利用混合类的话应该会大有好处。

作为一个专业的 C++ 程序员，你应当少花些时间做这些重复劳动，而应该把更多的时间用来考虑如何以新的方式实现可重用概念。以下技术都是一些通用的方法，可以直接在自己的程序中应用，也可以针对需要进行定制。

### 25.2.1 带引用计数的智能指针

第 4 章和第 13 章介绍了一个智能指针（smart pointer）的概念：这种方法是把动态分配的内存包装在一个安全的基于栈的变量中。第 16 章使用一个模板类提供了一个智能指针的实现。以下技术将增加引用计数，从而对第 16 章中的例子加以改进。

#### 引用计数的需要

作为一个一般概念，引用计数（reference counting）技术就是跟踪正在使用的一个类或特定对象的实例个数。引用计数智能指针会进行跟踪，即跟踪已经建立了多少个智能指针来引用某个实际指针（底层指针）。这样一来，智能指针就可以避免二次删除的情况。

如果是非引用计数智能指针，很容易导致二次删除问题。考虑以下类 Nothing，它只是在创建或撤销对象时打印消息。

```
class Nothing
{
public:
    Nothing() { cout << "Nothing::Nothing()" << endl; }
    ~Nothing() { cout << "Nothing::~Nothing()" << endl; }
};
```

如果要创建两个标准 C++ auto\_ptr，并让它们都指向同一个 Nothing 对象，两个智能指针出作用域时都会尝试删除同一个对象：

```
void doubleDelete()
{
    Nothing* myNothing = new Nothing();
```

```

    auto_ptr<Nothing*> autoPtr1(myNothing);
    auto_ptr<Nothing*> autoPtr2(myNothing);
}

```

上述函数的输出将是：

```

Nothing::Nothing()
Nothing::~~Nothing()
Nothing::~~Nothing()

```

奇怪了！只有一个构造函数调用，却有两个析构函数调用？而且本来这个类是为了让指针更安全的，怎么还会出现这种问题呢？

如果只是对一些简单情况使用智能指针，如分配仅在一个函数中使用的内存，那么这就不是问题。不过，如果程序在一个数据结构中存储了多个智能指针，或者会复制智能指针、对智能指针赋值，或将智能指针作为参数传递给函数，这就会使智能指针的使用复杂化，为此另外增加一层安全绝对是必要的。

引用计数智能指针比内置的 `auto_ptr` 更安全，因为它会跟踪指针的引用数，只在指针不再使用的情况下才会释放内存。

### SuperSmartPointer

`SuperSmartPointer` 是一个引用计数的智能指针实现，其方法是为引用计数维护一个静态的 `map`。`map` 中的每个键是一个传统指针的内存地址，可能有一个或多个 `SuperSmartPointer` 指示这个传统指针。与键对应的值是指示该对象的 `SuperSmartPointer` 的个数。

以下 `SuperSmartPointer` 的实现以第 16 章所示的智能指针代码为基础。在继续学习之前，你可能需要先回顾一下第 16 章的相应代码。在此主要变化出现在两种情况下，即设置一个新指针时（通过单参数构造函数、复制构造函数或 `operator=`），以及 `SuperSmartPointer` 用完一个底层指针时（通过撤销，或者用 `operator=` 重新赋值）。

初始化一个新指针时，`initPointer()` 方法会检查静态 `map`，查看该指针是否已经包含在一个既有的 `SuperSmartPointer` 中。倘非如此，计数则初始化为 1。如果确实已经在 `map` 中，计数则递增。指针重新赋值或者撤销外层 `SuperSmartPointer` 时，会调用 `finalizePointer()` 方法。如果 `map` 中没有发现这个指针，此方法首先会打印一个错误。如果找到了指针，计数会减 1。如果这导致计数减为 0，那么底层指针就可以安全地释放了。此时，会把键/值对显式地从 `map` 删除，使 `map` 的大小相应缩小。

```

#include <map>
#include <iostream>

template <typename T>
class SuperSmartPointer
{
public:
    explicit SuperSmartPointer(T* inPtr);
    ~SuperSmartPointer();

    SuperSmartPointer(const SuperSmartPointer<T>& src);
    SuperSmartPointer<T>& operator=(const SuperSmartPointer<T>& rhs);
    const T& operator*() const;
    const T* operator->() const;
    T& operator*();
    T* operator->();
};

```

```
operator void*() const { return mPtr; }

protected:
    T* mPtr;
    static std::map<T*, int> sRefCountMap;

    void finalizePointer();
    void initPointer(T* inPtr);
};

template <typename T>
std::map<T*, int> SuperSmartPointer<T>::sRefCountMap;

template <typename T>
SuperSmartPointer<T>::SuperSmartPointer(T* inPtr)
{
    initPointer(inPtr);
}

template <typename T>
SuperSmartPointer<T>::SuperSmartPointer(const SuperSmartPointer<T>& src)
{
    initPointer(src.mPtr);
}

template <typename T>
SuperSmartPointer<T>&
SuperSmartPointer<T>::operator=(const SuperSmartPointer<T>& rhs)
{
    if (this == &rhs) {
        return (*this);
    }
    finalizePointer();
    initPointer(rhs.mPtr);

    return (*this);
}

template <typename T>
SuperSmartPointer<T>::~SuperSmartPointer()
{
    finalizePointer();
}

template<typename T>
void SuperSmartPointer<T>::initPointer(T* inPtr)
{
    mPtr = inPtr;
    if (sRefCountMap.find(mPtr) == sRefCountMap.end()) {
        sRefCountMap[mPtr] = 1;
    } else {
        sRefCountMap[mPtr]++;
    }
}

template<typename T>
```

```

void SuperSmartPointer<T>::finalizePointer()
{
    if (sRefCountMap.find(mPtr) == sRefCountMap.end()) {
        std::cerr << "ERROR: Missing entry in map!" << std::endl;
        return;
    }
    sRefCountMap[mPtr]--;
    if (sRefCountMap[mPtr] == 0) {
        // No more references to this object--delete it and remove from map
        sRefCountMap.erase(mPtr);
        delete mPtr;
    }
}

template <typename T>
const T* SuperSmartPointer<T>::operator->() const
{
    return (mPtr);
}

template <typename T>
const T& SuperSmartPointer<T>::operator*() const
{
    return (*mPtr);
}

template <typename T>
T* SuperSmartPointer<T>::operator->()
{
    return (mPtr);
}

template <typename T>
T& SuperSmartPointer<T>::operator*()
{
    return (*mPtr);
}

```

### 对 SuperSmartPointer 单元测试

以上定义的 Nothing 类可以用于对 SuperSmartPointer 做一个简单的单元测试。需要做一个修改来确定测试是否通过。为此要向 Nothing 类增加两个静态成员，分别跟踪分配数和删除数。构造函数和析构函数会修改这些值，而不是打印一个消息。如果 SuperSmartPointer 工作正常，程序终止时，这两个数总是一样的。

```

class Nothing
{
public:
    Nothing() { sNumAllocations++; }
    ~Nothing() { sNumDeletions++; }

    static int sNumAllocations;
    static int sNumDeletions;
};

int Nothing::sNumAllocations = 0;
int Nothing::sNumDeletions = 0;

```

下面是具体的测试。需要注意，这里用了另外一组大括号来保证 SuperSmartPointer 在自己的作用域中，这样 SuperSmartPointer 的分配和撤销都发生在函数内部。

```
void testSuperSmartPointer()
{
    Nothing* myNothing = new Nothing();

    {
        SuperSmartPointer<Nothing> ptr1(myNothing);
        SuperSmartPointer<Nothing> ptr2(myNothing);
    }

    if (Nothing::sNumAllocations != Nothing::sNumDeletions) {
        std::cout << "TEST FAILED: " << Nothing::sNumAllocations <<
            " allocations and " << Nothing::sNumDeletions <<
            " deletions" << std::endl;
    } else {
        std::cout << "TEST PASSED" << std::endl;
    }
}
```

如果成功地执行了这个测试程序，会得到以下输出：

```
TEST PASSED
```

你还应当为 SuperSmartPointer 类编写另外的测试。例如，应当测试复制构造函数和 operator= 的功能。

#### 改进此实现

静态引用计数映射为 SuperSmartPointer 在内置的 C++ 智能指针之上又增加了一层安全。不过，这个新实现并非没有问题。

应该记得，这里对应每个类型都存在着一个模板。换句话说，如果有一些 SuperSmartPointer 存储整数指针，另外一些 SuperSmartPointer 存储字符指针，那么在编译时实际上会生成两个类：SuperSmartPointer<int> 和 SuperSmartPointer<char>。由于引用计数映射静态存储于类中，就会生成两个映射。在大多数情况下，这没有什么问题，不过可以把一个 char\* 强制转换为一个 int\*，这就会得到指示同一个变量的两个不同模板类的两个 SuperSmartPointer。由于表数据是单独的，这就会导致二次删除，如下代码所示。

```
char* ch = new char;

SuperSmartPointer<char> ptr1(ch);
SuperSmartPointer<int> ptr2((int*)ch); // BUG! Double deletion will occur!
```

对这个问题，一种解决方案是让引用映射作为一个全局变量，不过通常并不赞成采用全局变量。另一个解决方案是把映射包装在一个非模板类，可以把这个包装类叫做 MapManager，由 SuperSmartPointer 模板类引用。

这个实现还有一个问题，它不是线程安全的。你已经了解了，线程并不是 C++ 语言的一个特性。对静态映射的访问应该用一个锁来保护，这样并发的增加和删除就不会彼此冲突。

如果在成品代码中使用 SuperSmartPointer，就应该考虑以上给出的代码对应用是否合适，看看是否需要增加线程安全和一个全局映射。



### 25.2.2 双重分派

双重分派 (double dispatch) 技术为多态概念增加了一层含义。在第 3 章已经介绍过，多态允许程序基于运行时类型来确定行为。例如，可以有一个提供 `move()` 方法的 `Animal` 类。所有 `Animal` 都会移动，但是它们移动的方式不同。要为 `Animal` 的每个子类定义 `move()` 方法，这样就能在运行时对应适当的动物来调用或分派适当的方法，而无需在编译时就知道动物的类型。第 10 章解释了如何使用虚方法来实现这种运行时多态。

不过，有时需要一个方法按照两个对象的运行时类型做相应表现，而不只是基于一个对象的运行时类型。例如，假设为 `Animal` 类增加一个方法，如果这个动物捕食另一个动物，这个方法就返回 `true`，否则返回 `false`。这个决定要基于两个因素做出：捕食另一个动物的动物类型，以及被吃动物的类型。遗憾的是，C++ 没有提供相应的语言机制来基于多个对象的运行时类型选择一个行为。虚方法本身不足以对这种情况建模，因为它们只是基于接收到的一个对象的运行时类型来确定方法或行为。

有些面向对象语言提供了这种能力，可以基于两个或多个对象的运行时类型在运行时选择一个方法。它们称这种特性为多方法 (multi-method)。不过，在 C++ 中，并没有支持这种“多方法”的核心语言特性。幸运的是，双重分派技术可以让函数作为多个对象的虚函数（译者注：即可以根据多个对象的运行时类型决定选择何种行为）。

双重分派实际上是多重分派的一个特例，多重分派是指根据两个或多个对象的运行时类型来选择一个行为。双重分派则是基于两个对象的运行时类型选择一个行为，实际上，这就已经足够了。

#### 第 1 个尝试：强力方法

要实现一个方法，其行为根据两个不同对象的运行时类型来确定，为此最直接的方法是，从其中一个对象的角度出发，使用一系列 `if/else` 构造来检查另一个对象的类型。例如，可以在每个 `Animal` 子类上实现一个名为 `eats()` 的方法，它取另一个动物作为参数。这个方法要在基类中声明为纯虚方法，如下所示：

```
class Animal
{
public:
    virtual bool eats(const Animal& inPrey) const = 0;
};
```

每个子类都要实现 `eats()` 方法，并基于参数的类型返回适当的值。以下是多个子类 `eats()` 的实现。要注意 `Dinosaur` 子类没有使用一系列 `if/else` 构造，因为（在作者看来）恐龙什么都吃。

```
bool Bear::eats(const Animal& inPrey) const
{
    if (typeid(inPrey) == typeid(Bear&)) {
        return false;
    } else if (typeid(inPrey) == typeid(Fish&)) {
        return true;
    } else if (typeid(inPrey) == typeid(Dinosaur&)) {
        return false;
    }
    return false;
}

bool Fish::eats(const Animal& inPrey) const
```



```

{
    if (typeid(inPrey) == typeid(Bear&)) {
        return false;
    } else if (typeid(inPrey) == typeid(Fish&)) {
        return true;
    } else if (typeid(inPrey) == typeid(Dinosaur&)) {
        return false;
    }
    return false;
}

bool Dinosaur::eats(const Animal& inPrey) const
{
    return true;
}

```

强力方法是可行的，而且对于少量的类来说，这也是最直接的技术。不过，出于一些原因，你可能不想采用这种做法，具体原因包括：

- 在此要显式地查找对象的类型，追求纯粹 OOP 的人对此往往会持反对意见，因为这意味着设计缺乏一个合适的面向对象结构。
- 由于所有类型都在一个方法中检查，子类必须覆盖所有情况，或者不覆盖任何情况。例如，如果想实现一个 `CannibalisticBear` 类，它会捕食其他的 `Bear`，必须在子类中重新实现所有现有的 `Bear` 捕食行为。
- 随着类型的增多，这种代码会变得越来越混乱，而且重复性越来越高。
- 这种方法没有要求子类去考虑新类型。例如，如果增加了一个 `Donkey` 子类，`Bear` 类仍能编译通过，不过让它捕食一个 `Donkey` 时，会返回 `false`，尽管所有人都知道熊肯定是吃驴子的。

#### 第 2 个尝试：利用重载实现单向多态

你可能想利用重载来使用多态，从而避免所有层联的 `if/else` 构造。可以不为每个类都提供一个取 `Animal` 引用的 `eats()` 方法，为什么不让各个 `Animal` 子类重载这个方法呢？基类定义可能如下所示：

```

class Animal
{
public:
    virtual bool eats(const Bear& inPrey) const = 0;
    virtual bool eats(const Fish& inPrey) const = 0;
    virtual bool eats(const Dinosaur& inPrey) const = 0;
};

```

由于超类中的方法是纯虚方法，每个子类都必须为每种其他类型的 `Animal` 实现相应的行为。例如，`Bear` 类可能包含以下方法：

```

class Bear : public Animal
{
public:
    virtual bool eats(const Bear& inBear) const { return false; }
    virtual bool eats(const Fish& inFish) const { return true; }
    virtual bool eats(const Dinosaur& inDinosaur) const { return false; }
};

```

这个方法开始看上去是可行的，但是它实际上只解决了问题的一半。为了对一个 `Animal` 调用适当的 `eats()` 方法，编译器需要知道被吃动物的编译时类型。以下的调用会成功，因为捕食动物和被捕食动物的

编译时类型都是已知的：

```
Bear myBear;
Fish myFish;

cout << myBear.eats(myFish) << endl;
```

这个解决方案中没有解决的问题是：只有一个方向的多态。可以在一个 `Animal` 的上下文中访问 `myBear`，而且会调用正确的方法：

```
Bear myBear;
Fish myFish;
Animal& animalRef = myBear;

cout << animalRef.eats(myFish) << endl;
```

反过来则不成立。如果在 `Animal` 类的上下文中访问 `myFish`，并将其传递至（另一个动物的）捕食方法，就会得到一个编译错误，因为并没有取 `Animal` 作为参数的捕食方法。编译器无法在编译时确定要调用哪个版本。以下例子就无法编译通过：

```
Bear myBear;
Fish myFish;
Animal& animalRef = myFish;

cout << myBear.eats(animalRef) << endl; // BUG! No such method Bear::eats(Animal&)
```

由于编译器要在编译时知道将调用哪个重载版本的 `eats()` 方法，所以这种解决方案并不是真正的多态。例如，如果要迭代处理一个 `Animal` 引用数组，并把各 `Animal` 引用传递至一个 `eats()` 调用，这是无法做到的。

### 第3个尝试：双重分派

双重分派技术是多类型问题的一个真正多态的解决方案。在 C++ 中，多态是通过在子类中覆盖方法获得的。在运行时，会基于对象的实际类型来调用方法。以上的单向多态方法不可行，原因是它想使用多态来确定要调用哪个重载版本的方法，而不是使用多态确定哪个类来调用方法。

我们先来看一个子类，可以考虑 `Bear` 类。这个类需要一个有以下声明的方法：

```
virtual bool eats(const Animal& inPrey) const;
```

双重分派的关键是基于对参数的方法调用来确定结果。假设 `Animal` 类有一个名为 `eatenBy()` 的方法，它取 `Animal` 引用作为一个参数。如果当前 `Animal` 能被传入的动物吃掉，此方法就返回 `true`。有了这个方法，`eats()` 的定义就变得非常简单：

```
bool Bear::eats(const Animal& inPrey) const
{
    return inPrey.eatenBy(*this);
}
```

首先，这种解决方案看上去只是向单向多态方法增加了另外一层方法调用。毕竟，各个子类仍然必须为 `Animal` 的每个子类实现一个版本的 `eatenBy()`。不过，在此有一个重大区别。多态出现了两次！对 `Animal` 调用捕食方法时，多态可以确定你是在调用 `Bear::eats`、`Fish::eats` 还是其他某个版本的 `eats` 方法。调用 `eatenBy()` 时，多态又会确定要调用哪个类的 `eatenBy()` 方法。它要基于 `inPrey` 对象的运行时类

型调用 `eatenBy()`。注意，`*this` 的运行时类型总是与编译时类型相同，这样编译器就可以针对参数（在此例中为 `Bear`）正确地调用重载版本的 `eatenBy()`。

以下是使用双重分派的 `Animal` 层次体系的类定义。注意，这里的前置类声明是必要的，因为基类使用了子类的引用。

```
// forward references
class Fish;
class Bear;
class Dinosaur;

class Animal
{
public:
    virtual bool eats(const Animal& inPrey) const = 0;

    virtual bool eatenBy(const Bear& inBear) const = 0;
    virtual bool eatenBy(const Fish& inFish) const = 0;
    virtual bool eatenBy(const Dinosaur& inDinosaur) const = 0;
};

class Bear : public Animal
{
public:
    virtual bool eats(const Animal& inPrey) const;

    virtual bool eatenBy(const Bear& inBear) const;
    virtual bool eatenBy(const Fish& inFish) const;
    virtual bool eatenBy(const Dinosaur& inDinosaur) const;
};

class Fish : public Animal
{
public:
    virtual bool eats(const Animal& inPrey) const;

    virtual bool eatenBy(const Bear& inBear) const;
    virtual bool eatenBy(const Fish& inFish) const;
    virtual bool eatenBy(const Dinosaur& inDinosaur) const;
};

class Dinosaur : public Animal
{
public:
    virtual bool eats(const Animal& inPrey) const;

    virtual bool eatenBy(const Bear& inBear) const;
    virtual bool eatenBy(const Fish& inFish) const;
    virtual bool eatenBy(const Dinosaur& inDinosaur) const;
};
```

以下是实现。注意，每个 `Animal` 子类都以同样的方式实现了 `eats()` 方法，但是不能将其向上重构到父类中。原因是，如果你打算这么做，编译器就无法知道要调用哪个重载版本的 `eatenBy()` 方法，因为 `*this` 将是一个 `Animal`，而不是一个特定的子类。应该记得，方法重载解析是根据对象的编译时类型确定的，而不是根据运行时类型。

```
bool Bear::eats(const Animal& inPrey) const
{
    return inPrey.eatenBy(*this);
}

bool Bear::eatenBy(const Bear& inBear) const
{
    return false;
}

bool Bear::eatenBy(const Fish& inFish) const
{
    return false;
}

bool Bear::eatenBy(const Dinosaur& inDinosaur) const
{
    return true;
}

bool Fish::eats(const Animal& inPrey) const
{
    return inPrey.eatenBy(*this);
}

bool Fish::eatenBy(const Bear& inBear) const
{
    return true;
}

bool Fish::eatenBy(const Fish& inFish) const
{
    return true;
}

bool Fish::eatenBy(const Dinosaur& inDinosaur) const
{
    return true;
}

bool Dinosaur::eats(const Animal& inPrey) const
{
    return inPrey.eatenBy(*this);
}

bool Dinosaur::eatenBy(const Bear& inBear) const
{
    return false;
}

bool Dinosaur::eatenBy(const Fish& inFish) const
{
    return true;
}
```

```
    return false;
}

bool Dinosaur::eatenBy(const Dinosaur& inDinosaur) const
{
    return true;
}
```

双重分派概念刚开始时可能不太容易适应。建议多看看这个代码，充分理解这个概念及其实现。

### 25.2.3 混合类

第 3 章和第 10 章介绍了一种使用多重继承构建混合类 (mix-in class) 的技术。应该记得，混合类为一个既有层次体系中的类增加了一小部分行为。通常根据类名就可以把混合类识别出来，如 Clickable、Drawable、Printable 或 Lovable。

#### 设计一个混合类

混合类有多种形式。由于混合类并不是一个正式的语言特性，可以采用希望的任何形式编写混合类，而不会违反任何规则。有些混合类只是指示一个类支持某个行为，如假想有一个 Drawable 混合类。混合 Drawable 类的所有类都必须实现方法 draw()。这个混合类本身不包含任何功能，它只是把一个对象标志为支持 draw() 行为。这种用法类似于 Java 中的接口概念，Java 接口就是一组预定义的行为，但是不提供具体实现。

其他混合类可能包含有具体的代码。可以有一个名为 Playable 的混合类，这个类将混合到某些类型的媒体对象中。混合类可以包含与计算机声音驱动程序通信的大部分代码。通过混合这个类，媒体对象就能够很自然地获得这个功能。

在设计一个混合类时，需要考虑你要增加什么行为，还要考虑这个行为是放在对象层次体系中，还是放在一个单独的类里。还是用前面的例子，如果所有媒体类都是可播放的，基类就应该从 Playable 派生，而不是把 Playable 类混合到所有子类中。如果只有某些媒体类是可播放的，而且它们分布在层次体系中的多个位置上，就应该考虑采用混合类。

有些情况下混合类尤其有用，如果你的类按一个轴组织为一个层次体系，但是这些类在另一个轴上也可能存在相似性，这就属于此类情况。例如，考虑一个网格战争模拟游戏。每个网格位置可以包含一个 Item，它有攻击和防御能力以及其他一些特性。有些项（如 Castle）是静止的。另外一些（如 Knight 或 FloatingCastle）可以在网格上移动。最初设计对象层次体系时，可能会得到如图 25-1 的设计，在此按照各个类的攻击和防御能力来组织。

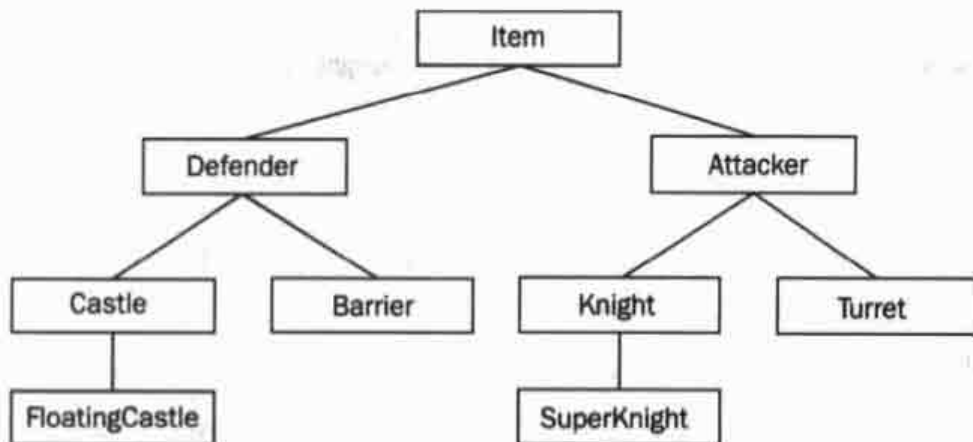


图 25-1



图 25-1 中的层次体系没有考虑移动功能，但某些类有此功能。基于移动功能来建立层次体系会得到如图 25-2 所示的结构。

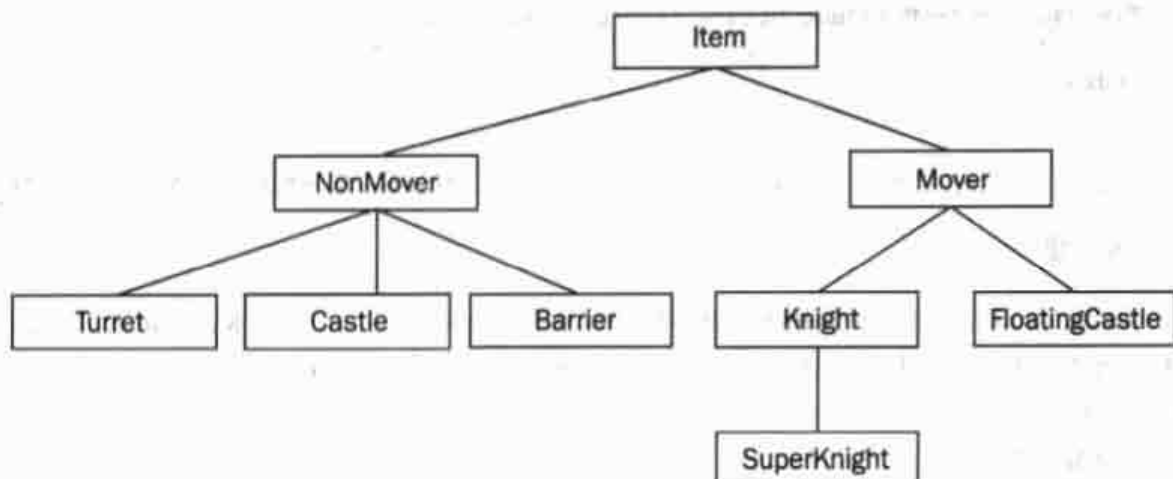


图 25-2

当然，图 25-2 的设计完全把图 25-1 的组织抛在一边。好的面向对象程序员要怎么做呢？

对于这个问题，有两个常用的解决方案。假设以第一个层次为主，即按攻击项和防御项来组织，就需要某种方式把移动情况考虑进来。一种可能是，即使只有一部分子类支持移动，还是可以为 Item 基类增加一个 move() 方法。此方法的默认实现什么也不做。某些子类可以覆盖 move() 来具体改变其在网格上的位置。

另一种方法是编写一个 Movable 混合类。图 25-1 的层次体系还是不错的，可以保留这个体系，不过其中的某些类除了派生图中所示的父类外，还要派生 Movable。图 25-3 显示了这个设计。

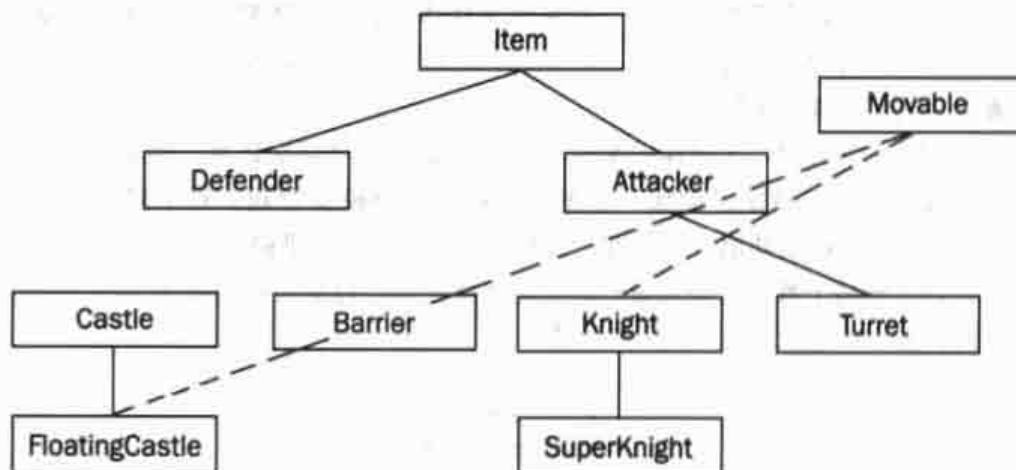


图 25-3

### 实现一个混合类

编写混合类与编写一个正常的类并没有什么区别。实际上，混合类的编写更为简单。还是看前面的战争模拟游戏，Movable 混合类可以如下所示：

```

class Movable
{
public:
    virtual void move() = 0;
};
  
```



如前面所定义的, Movable 混合类并没有包含任何具体的功能。不过, 它做了两个非常重要的事情。首先, 它为可以移动的 Item 提供了一个类型。这样就能针对所有可移动的项做些工作, 如创建所有可移动项的一个列表, 而无需知道也不用关心各项属于 Item 的哪个具体子类。Movable 类还声明了所有可移动的项都必须实现一个名为 move() 的方法。这样一来, 可以迭代处理所有 Movable 对象, 并让它们移动。

### 使用混合类

使用混合类的代码与多重继承的代码在语法上是相同的。除了派生主层次体系中的父类外, 还要派生混合类:

```
class FloatingCastle : public Castle, public Movable
{
    public:
        virtual void move();
    // Other methods and members not shown here
}
```

剩下的任务就是为 FloatingCastle 提供 move() 方法的一个定义。完成之后, 就得到了这样一个类, 它不仅位于层次体系中最合理的位置上, 而且还与层次体系中其他位置上的对象享有某种共同性。

## 25.3 面向对象框架

20 世纪 80 年代图形操作系统首次露面时, 过程性程序设计是当时的标准。那时, 编写一个 GUI 应用通常涉及处理复杂的数据结构, 还要将其传递至操作系统提供的函数。例如, 为了在窗口上绘制一个矩形, 可以用适当的信息填写一个 Window 结构, 并把它传递给一个 drawRect() 函数。

随着面向对象程序设计越来越流行, 程序员也开始寻求合适的途径在 GUI 开发中应用 OO 模式。这就得到了所谓的面向对象框架 (Object-Oriented Framework)。一般来说, 框架就是一组类, 这些类共同使用来为某些底层功能提供面向对象的接口。谈到框架时, 程序员通常是指用于一般应用开发的大型类库。不过, 框架实际上可以表示任意规模的功能。如果你写了一组类, 这些类可以为应用提供数据库功能, 那么它们也可以认为是一个框架。

### 25.3.1 使用框架

框架的关键特性是它会提供自己的一组技术和模式。开始使用框架时, 通常需要先有所学习, 因为框架总有自己的思想模型。在使用一个大型应用框架之前, 如 Microsoft Foundation Classes (MFC), 需要了解它的世界观。

在抽象思想和具体实现方面, 框架之间可能大相径庭。许多框架都建立在传统过程性 API 之上, 这可能会对设计的许多方面产生影响。另外一些框架则完全本着面向对象设计思想来建立。有些框架可能执意反对 C++ 语言中的某些方面, 如 BeOS 框架就对多重继承概念很不以为然。

开始使用一个新框架时, 第一个任务是要了解如何让它运作起来。它采用何种设计原则? 这个框架的开发人员想反映什么样的思想模型? 它大量使用了语言的哪些方面? 这些都是很重要的问题, 尽管听上去这些问题好像在使用框架的过程中也能明确。但是, 如果不能理解框架的设计、模型或语言特性, 你很快就会越出框架的合理界限。例如, 如果框架使用了一个定制 String 类, 而你用 C 风格的字符串编写代码, 这就要求做大量转换工作, 你很快就会被这些转换工作搞得昏头转向, 而这本来是可以避免的。

如果能理解框架的设计, 还有可能对其扩展。例如, 如果框架忽略了某个特性, 如没有对打印提供支持, 就可以用框架所用的同一模型来编写自己的打印类。这么一来, 应用就能保持一个一致的模型,

而且你的代码也可以得到其他应用的重用。

### 25.3.2 模型-视图-控制器模式

前面已经提到，不同框架在实现面向对象设计的方法上会有所不同。有一种常用的模式称为模型-视图-控制器（model-view-controller，MVC）。许多应用通常都会处理一组数据、这组数据之上的一个或多个视图，以及对数据的操作。MVC 正是对这个概念的建模。

在 MVC 中，模型（model）是指一组数据。在一个赛车仿真器中，模型会记录各种统计数据，如当前车速和承受的破坏程度。在实际中，模型可能以一个类的形式出现（带有多个获取方法和设置方法）。赛车模型的类定义可以如下所示：

```
class RaceCar
{
    public:
        RaceCar();

        int getSpeed();
        void setSpeed(int inValue);

        int getDamageLevel();
        void setDamageLevel(int inValue);

    protected:
        int mSpeed;
        int mDamageLevel;
};
```

视图（view）是对模型的一个特定视图表示。例如，RaceCar 可以有两个视图。第一个视图可以是车的一个图形视图，第二个可能是一个图表来显示破坏程度随时间的变化情况。关键在于这两个视图都在相同的数据上操作，它们的区别是查看相同信息的方式有所不同。这正是 MVC 模式的主要优点之一，通过将数据与显示相分离，代码可以得到更好的组织，而且很容易创建另外的视图。

MVC 模式的最后一部分是控制器（controller）。控制器就是修改模型从而对某个事件做出响应的一段代码。例如，当赛车仿真器的驾驶员碰上一个实物障碍，控制器就会指导模型增加其破坏程度，并减速。

MVC 的三个组件以一种反馈环的形式交互。动作由控制器处理，这会调整模型，导致视图改变。这个交互如图 25-4 所示。

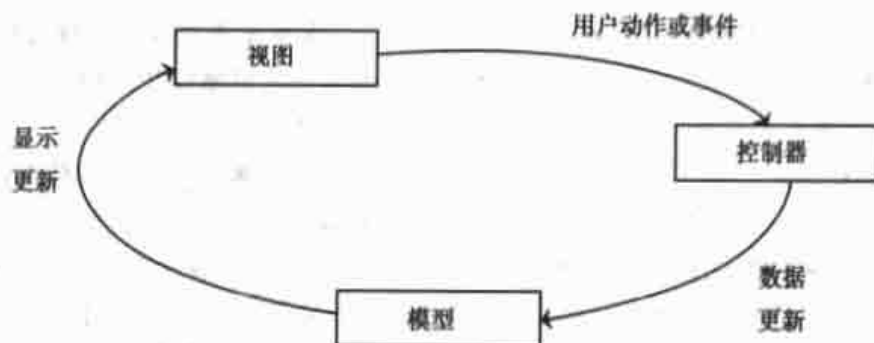


图 25-4

模型-视图-控制器模式在许多流行的框架中都得到了广泛的支持。即使是非传统应用（如 Web 应用）也开始朝着 MVC 的方向发展，因为它清楚地分离了数据、数据的操作和数据的显示。

## 25.4 小结

在本章中，你了解了一些常用的技术，专业的 C++ 程序员经常在项目中使用这些技术。在你逐步成为一个软件开发人员的过程中，肯定已经建立了自己的可重用类和库集。如果能发现设计技术，这就为开发和使用模式（pattern）打开了一扇大门，模式是更高级的可重用构造。第 26 章将介绍模式的一些应用。

## 第 26 章 应用设计模式

设计模式概念是一种很简单但同时又很有影响力的思想。如果你能发现程序中反复出现的面向对象交互，要找出一个完美的解决方案，实际上只是要选择可以应用哪一种合适的模式。在这本书里已经深入地谈到了一种模式，即迭代器模式，STL 中就大量使用了这种模式。这一章将详细介绍另外的一些设计模式，并提供一些示例实现。

第 4 章中我们介绍过，设计模式是一个新兴的概念。某些模式会以不同的名字出现，或者有不同的解释。程序员之间似乎对设计的每一个方面都会存在争议，在我们看来，这是一个好事情。不要只是接受这些模式，把它们做为完成某个任务的惟一途径，而应当仔细分析这些模式的方法和思想，对其改进，建立新的模式。

本章讨论了以下模式：

- 单例模式 (Singleton)
- 工厂模式 (Factory)
- 代理模式 (Proxy)
- 适配器模式 (Adapter)
- 装饰器模式 (Decorator)
- 职责链模式 (Chain of Responsibility)
- 观察者/监听者模式 (Observer/Listener)

### 26.1 单例模式

单例 (singleton) 是最简单的设计模式之一。在英语中，“singleton”一词表示“一类中的一个”或“单个”。在程序设计中，这个词的含义也是类似的。单例模式作为一种策略，要求一个程序中只能存在一个类的一个实例。对类应用单例模式可以保证只会创建这个类的一个对象。单例模式还指出可以从程序中的任何位置全局访问这样一个对象。程序员通常把遵循单例模式的类称为单例类 (singleton class) (译者注：作者后面甚至把单例类干脆简写作单例)。

设计一个程序时，如果有一个通用的应用类 (application class)，它要处理应用的启动、关闭和流程控制，此时单例模式就特别有帮助。一个程序中有两个应用对象往往是不合适的。实际上，如果有两个应用对象，它们都认为自己在控制着应用的流程，这可能会带来灾难。通过使用单例模式，可以确保从程序中的任何位置只能访问惟一的一个应用对象。

只要想在程序中只创建一个类的一个对象，就应该使用单例模式。如果程序有一个前提假设，认为某个类只有一个实例，就应该利用单例模式来保证这个假设。

#### 26.1.1 举例：日志机制

单例对于工具类特别有用。许多应用都有一个日志工具的概念，日志工具就是一个类，它要负责把



```

// Logs a vector of messages at the given log level
static void log(const std::vector<std::string>& inMessages,
               const std::string& inLogLevel);

// Closes the log file
static void teardown();

protected:
    static void init();

    static const char* const kLogFileName;

    static bool sInitialized;
    static std::ofstream sOutputStream;

private:
    Logger() {}
};

```

Logger 类的实现相当简单。会在每一个日志调用中检查 sInitialized 静态成员，从而确保已经调用了 init() 方法来打开日志文件。一旦日志文件已经打开，每个日志消息都会写至此文件，并附带相应的日志级别。

```

/**
 * Logger.cpp
 *
 * Implementation of a singleton logger class
 */
#include <string>
#include "Logger.h"

using namespace std;

const string Logger::kLogLevelDebug = "DEBUG";
const string Logger::kLogLevelInfo = "INFO";
const string Logger::kLogLevelError = "ERROR";

const char* const Logger::kLogFileName = "log.out";

bool Logger::sInitialized = false;
ofstream Logger::sOutputStream;

void Logger::log(const string& inMessage, const string& inLogLevel)
{
    if (!sInitialized) {
        init();
    }
    // Print the message and flush the stream with endl.
    sOutputStream << inLogLevel << ": " << inMessage << endl;
}

void Logger::log(const vector<string>& inMessages, const string& inLogLevel)
{
    for (size_t i = 0; i < inMessages.size(); i++) {

```



```

        log(inMessages[i], inLogLevel);
    }
}

void Logger::teardown()
{
    if (sInitialized) {
        sOutputStream.close();
        sInitialized = false;
    }
}

void Logger::init()
{
    if (!sInitialized) {
        sOutputStream.open(kLogFileName, ios_base::app);
        if (!sOutputStream.good()) {
            cerr << "Unable to initialize the Logger!" << endl;
            return;
        }
        sInitialized = true;
    }
}
}

```

### 访问控制单例

热衷于面向对象的人（警告：这样的人到处都是，可能就在你的公司中！）可能对单例问题的静态类解决方案很不以为然。由于无法实例化一个 Logger 对象，也就无法建立一个日志工具层次体系，更不能利用多态。尽管这种层次体系在单例情况下很少用，但这确实是一个缺点。也许更重要的是，由于完全使用静态方法，这会导致根本没有面向对象。前一个例子中建立的类实际上只是一个 C 风格函数的集合，而不是一个内聚性的类。

要用 C++ 建立一个真正的单例，可以在使用 static 关键字的同时还使用访问控制机制。采用这种方法，在运行时就会存在一个真正的 Logger 对象，而且类可以保证只会存在一个对象。客户总能通过一个名为 instance() 的静态方法得到这个对象。这个类定义如下所示：

```

/**
 * Logger.h
 *
 * Definition of a true singleton logger class
 */
#include <iostream>
#include <fstream>
#include <vector>

class Logger
{
public:
    static const std::string kLogLevelDebug;
    static const std::string kLogLevelInfo;
    static const std::string kLogLevelError;

    // Returns a reference to the singleton Logger object
    static Logger& instance();

    // Logs a single message at the given log level

```

```

    void log(const std::string& inMessage,
             const std::string& inLogLevel);

    // Logs a vector of messages at the given log level
    void log(const std::vector<std::string>& inMessages,
             const std::string& inLogLevel);

protected:
    // Static variable for the one-and-only instance
    static Logger sInstance;

    // Constant for the filename
    static const char* const kLogFileName;

    // Data member for the output stream
    std::ofstream mOutputStream;

private:
    Logger();
    ~Logger();
};

```

已经可以很明显地看出这个方法的一个优点了。由于会存在一个具体的对象，原来在静态解决方案中提供的 `init()` 和 `teardown()` 方法现在分别代之以一个构造函数和一个析构函数。这是一个很大的改进，因为以前的静态解决方案要求客户显式地调用 `teardown()` 才能关闭文件。既然日志工具是一个对象，对象撤销时就可以关闭文件，这在程序结束时就会发生。

以下是相应实现。需要注意具体的 `log()` 方法仍保持不变，只不过它们不再是静态的了。构造函数和析构函数会自动得到调用，因为这个类包含了自身的一个实例作为一个静态成员。由于它们是私有的，因此外部代码无法创建或删除一个 `Logger`。

```

/**
 * Logger.cpp
 *
 * Implementation of a singleton logger class
 */
#include <string>
#include "Logger.h"

using namespace std;

const string Logger::kLogLevelDebug = "DEBUG";
const string Logger::kLogLevelInfo = "INFO";
const string Logger::kLogLevelError = "ERROR";

const char* const Logger::kLogFileName = "log.out";

// The static instance will be constructed when the program starts and
// destroyed when it ends.
Logger Logger::sInstance;

Logger& Logger::instance()
{
    return sInstance;
}

```

```
}
Logger::~Logger()
{
    mOutputStream.close();
}

Logger::Logger()
{
    mOutputStream.open(kLogFileName, ios_base::app);
    if (!mOutputStream.good()) {
        cerr << "Unable to initialize the Logger!" << endl;
    }
}

void Logger::log(const string& inMessage, const string& inLogLevel)
{
    mOutputStream << inLogLevel << ": " << inMessage << endl;
}

void Logger::log(const vector<string>& inMessages, const string& inLogLevel)
{
    for (size_t i = 0; i < inMessages.size(); i++) {
        log(inMessages[i], inLogLevel);
    }
}
```

### 26.1.3 使用单例

以下两个程序显示了上述两个不同版本 Logger 类的使用。

```
// TestStaticLogger.cpp

#include "Logger.h"
#include <vector>
#include <string>

int main(int argc, char** argv)
{
    Logger::log("test message", Logger::kLogLevelDebug);

    vector<string> items;
    items.push_back("item1");
    items.push_back("item2");

    Logger::log(items, Logger::kLogLevelError);

    Logger::teardown();
}
```

```
// TestTrueSingletonLogger.cpp

#include "Logger.h"
#include <vector>
#include <string>
```

```
int main(int argc, char** argv)
{
    Logger::instance().log("test message", Logger::kLogLevelDebug);

    vector<string> items;

    items.push_back("item1");
    items.push_back("item2");

    Logger::instance().log(items, Logger::kLogLevelError);
}
```

这两个程序功能都一样。执行之后，文件 log.out 应当包含以下行：

```
DEBUG: test message
ERROR: item1
ERROR: item2
```

## 26.2 工厂模式

实际生活中的工厂会制造实实在在的对象，比如桌子或汽车。类似地，面向对象编程中的工厂（factory）也会生产对象。在程序中使用工厂时，想要创建某个特定对象的代码段会向工厂请求一个对象实例，而不是自行调用对象构造函数。例如，一个室内装修程序可能有一个 FurnitureFactory 对象。部分代码需要一件家具时，它会调用 FurnitureFactory 对象的 createTable() 方法，这个方法则会返回一张新桌子。

乍一看，工厂似乎只会导致复杂的设计，而没有明显的好处。看起来你只是向程序增加了一层复杂性。如果不在 FurnitureFactory 上调用 createTable()，完全可以简单地直接创建一个新的 Table 对象。不过，工厂确实很有用。你不必在整个程序中到处遍布着创建各个对象的代码，而是可以把对象的创建集中在某一个特定的范围内。这样的集中放置通常是创建对象的一种更好的模型。

工厂的另一个好处是，可以按照类层次体系构造对象，而无需知道具体的类。从下面的例子可以看到，工厂可能与类层次体系是并行的。

### 26.2.1 举例：汽车工厂模拟

在真实世界中，当谈到开车时，可能不用专指某一种特定类型的汽车。你说的可能是一辆丰田车也可能是一辆福特车。这都没有关系，因为无论是丰田也好福特也好，都能开。下面假设你想要一辆新车，就必须指定你想要的是丰田还是福特，是这样吗？不尽然。你可以只是说“我想要辆车”，根据你身在何处，就可以得到一辆特定的车。如果你在一家丰田工厂里说“我想要辆车”，当然会得到一辆丰田车。（也可能会把你抓起来，这要看你是不是想去打劫）。如果你在一家福特厂说“我想要辆车”，得到的自然就是福特车。

这种概念同样适用于 C++ 程序设计。第一个概念是，一般的汽车都能开，这没有什么新颖之处。这是标准的多态，在第 3 章已经了解了。可以编写一个抽象的 Car 类，它定义了一个 drive() 方法。Toyota 和 Ford 都可以作为 Car 类的子类，如图 26-1 所示。

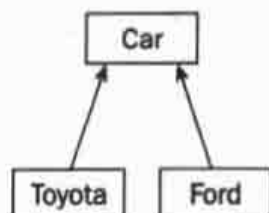


图 26-1

你的程序可以开 Car，而无须知道它究竟是 Toyota 还是 Ford。不过，利用标准面向对象程序设计，只需在一个地方指定 Toyota 或 Ford，这就是在创建汽车的时候指定。在此，需要调用某种车的构造函数。你不能只是说“我想要辆车”。不过，假设你已经有一个并行的汽车工厂类层次体系。CarFactory 超类可能定义了一个虚方法 buildCar()。ToyotaFactory 和 FordFactory 子类会覆盖 buildCar() 方法来构造一个 Toyota 或 Ford。图 26-2 显示了 CarFactory 层次体系。

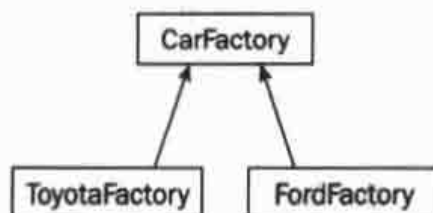


图 26-2

下面，假设程序中有一个 CarFactory 对象。当程序中的某段代码（如汽车经销商）想要一辆新车时，它会在 CarFactory 对象上调用 buildCar()。取决于这个汽车工厂究竟是一个 ToyotaFactory 还是一个 FordFactory，代码会得到一个 Toyota 或 Ford。图 26-3 显示了使用 ToyotaFactory 的汽车经销商程序中的对象。



图 26-3

图 26-4 显示了同一个程序，不过这一回是一个 FordFactory 而不是一个 ToyotaFactory。要注意 CarDealer 对象及其与工厂的关系仍保持不变。



图 26-4



这种方法的主要好处在于工厂抽象出了对象创建过程：可以在程序中很轻松地替换一个不同的工厂。只要能对所创建的对象使用多态，就可以对工厂使用多态：向汽车工厂请求一辆车时，你可能不知道这是一个丰田厂还是福特厂，不过不管怎样，它都会给你一辆能开的 Car。利用这种方法，可以得到一个能轻松扩展的程序：只需改变工厂实例，就可以让程序处理完全不同的一组对象和类。

### 26.2.2 实现工厂

使用工厂的一个原因是，你想创建的对象类型可能要取决于某种条件。例如，如果你是一个汽车经销商，想马上就要一辆车，你可能想直接向订单最少的工厂下订单，而不论最后得到的车是一个 Toyota 还是一个 Ford。以下实现将展示如何用 C++ 编写这样的工厂。

首先需要的是汽车的层次体系。为了使例子尽量简单，Car 类只有一个抽象方法，它会返回汽车的一个描述。Car 的两个子类也在以下例子中定义，在此使用了内联方法来返回描述。

```
/**
 * Car.h
 *
 */

#include <iostream>

class Car
{
public:
    virtual void info() = 0;
};

class Ford : public Car
{
public:
    virtual void info() { std::cout << "Ford" << std::endl; }
};

class Toyota : public Car
{
public:
    virtual void info() { std::cout << "Toyota" << std::endl; }
};
```

CarFactory 基类更有意思。每个工厂都会记录生产线上的汽车数。当调用公共的 requestCar() 方法时，工厂生产线上的汽车数就会增 1，调用纯虚方法 createCar() 会返回一辆新车。基本思想是，每个工厂都会覆盖 createCar() 来返回适当类型的车。工厂本身实现了 requestCar()，它会负责更新生产线上的汽车数。CarFactory 还提供了一个公共方法来查询每个工厂正在生产的汽车数。

CarFactory 的子类的类定义如下所示：



```
/**
 * CarFactory.h
 */

// For this example, the Car class is assumed to already exist.
#include "Car.h"

class CarFactory
{
public:
    CarFactory();

    Car* requestCar();

    int getNumCarsInProduction() const;

protected:
    virtual Car* createCar() = 0;

private:
    int mNumCarsInProduction;
};

class FordFactory : public CarFactory
{
protected:
    virtual Car* createCar();
};

class ToyotaFactory : public CarFactory
{
protected:
    virtual Car* createCar();
};
```

可以看到，子类只是覆盖 `createCar()` 来返回它们生产的特定类型的汽车。CarFactory 层次体系的实现如下所示：

```
/**
 * CarFactory.cpp
 */

#include "CarFactory.h"

// Initialize the count to zero when the factory is created.
CarFactory::CarFactory() : mNumCarsInProduction(0) {}

// Increment the number of cars in production and return the
// new car.
Car* CarFactory::requestCar()
{
    mNumCarsInProduction++;
    return createCar();
}

int CarFactory::getNumCarsInProduction() const
```

```

{
    return mNumCarsInProduction;
}

Car* FordFactory::createCar()
{
    return new Ford();
}

Car* ToyotaFactory::createCar()
{
    return new Toyota();
}

```

这个例子中所用的实现方法称为抽象工厂 (abstract factory)，因为所创建对象的类型取决于所用工厂类的具体 (concrete) 子类。可以在一个类而不是一个类层次体系中实现类似的模式。在这种情况下，会有一个 create() 方法取一个类或字符串参数，由此参数来确定要创建哪一个对象。例如，CarFactory 对象可以提供一个 buildCar() 方法，这个方法取一个表示汽车类型的字符串为参数，并构造相应的类型。不过，这种技术没有前一种有意思，而且灵活性也不如前述的工厂体系方法。

工厂方法是实现虚构造函数方法（可以创建不同类型的对象）的一个途径。例如，buildCar() 方法可以创建 Toyota 和 Ford，这取决于在何种具体工厂对象上调用此方法。

### 26.2.3 使用工厂

使用工厂最简单的方法就是对其实例化，并调用适当的方法，如以下代码段所示。

```

ToyotaFactory myFactory;

Car* myCar = myFactory.requestCar();

```

还有一个更有意思的例子，即充分利用虚构造函数思想，在生产线上汽车最少的工厂里制造汽车。为此需要有一个函数来查看多个工厂，并选择最不忙的工厂，如以下函数：

```

CarFactory* getLeastBusyFactory(const vector<CarFactory*>& inFactories)
{
    if (inFactories.size() == 0) return NULL;

    CarFactory* bestSoFar = inFactories[0];
    for (size_t i = 1; i < inFactories.size(); i++)
    {
        if (inFactories[i]->getNumCarsInProduction() <
            bestSoFar->getNumCarsInProduction()) {
            bestSoFar = inFactories[i];
        }
    }
    return bestSoFar;
}

```

以下示例程序利用了这个函数，在当前最不忙的工厂制造 10 辆车，而不论是什么品牌。

```

int main(int argc, char** argv)
{
    vector<CarFactory*> factories;

    // Create 3 Ford factories and 1 Toyota factory.
    FordFactory* factory1 = new FordFactory();
    FordFactory* factory2 = new FordFactory();
    FordFactory* factory3 = new FordFactory();
    ToyotaFactory* factory4 = new ToyotaFactory();

    // To get more interesting results, preorder some cars.
    factory1->requestCar();
    factory1->requestCar();
    factory2->requestCar();
    factory4->requestCar();

    // Add the factories to a vector.
    factories.push_back(factory1);
    factories.push_back(factory2);
    factories.push_back(factory3);
    factories.push_back(factory4);

    // Build 10 cars from the least busy factory.
    for (int i = 0; i < 10; i++) {
        CarFactory* currentFactory = getLeastBusyFactory(factories);
        Car* theCar = currentFactory->requestCar();
        theCar->info();
    }
}

```

执行这段程序时，会打印出所生产的各辆汽车的品牌。

```

Ford
Ford
Ford
Toyota
Ford
Ford
Ford
Toyota
Ford
Ford

```

这个结果并不出所料，因为在此以循环调度方式有效地对工厂进行了循环处理。不过，可以想像这样一种情况，多个经销商都有汽车需求，这样每个工厂的当前状态就不那么可预知了。

#### 26.2.4 工厂的其他使用

工厂模式不光能用于对实际的工厂建模，还可以有其他用途。例如，请考虑一个字处理器，希望它能支持多种语言的文档，每个文档都使用一种语言。在这个字处理器中，文档语言的选择需要不同的支持，这反映在许多不同的方面：文档中使用的字符集（是否需要加重字符），拼法检查工具、辞典以及显示文档的方式，等等。可以使用工厂来设计一个简洁的字处理器，即编写一个抽象 LanguageFactory 超类，并为感兴趣的语言建立具体的工厂（子类），如 EnglishLanguageFactory 和

FrenchLanguageFactory。用户指定一个文档的语言时，程序会实例化相应的语言工厂，并把它与文档关联起来。

在此之后，程序不需要知道文档中支持哪一个语言。需要某个与语言有关的功能时，只需请求 LanguageFactory。例如，需要一个拼法检查工具时，可以在工厂上调用 createSpellchecker() 方法，它会以适当的语言返回一个拼法检查工具。

## 26.3 代理模式

代理 (proxy) 模式是将类的抽象与底层表示相分离的模式之一。代理对象相当于一个实际对象的代表。如果使用实际对象很耗费时间或者根本不可能，在这些情况下，这种代理对象相当常用。

你可能已经使用过代理模式，只是自己没有正式地知道这一点。代理在单元测试中非常方便。不必使用真实的股价数据来测试一个股票预测工具，只需编写一个代理类，由它模拟股票输入的行为，不过使用的只是固定数据。

### 26.3.1 举例：隐藏网络连通性问题

考虑一个网络游戏，有一个 Player 类表示 Internet 上的一个参加游戏的人。Player 类会包括一些需要网络连通性的功能，如即时消息特性。如果玩家的连接变得很慢或者半天没有响应，表示这个人的 Player 对象就无法接收到即时消息。

因为不希望把网络问题暴露给用户，可能需要一个单独的类隐藏 Player 的网络部分。这个 PlayerProxy 对象会取代真正的 Player 的对象。这个类的客户可能一直使用 PlayerProxy 类作为实际 Player 类的一个监管人，或者系统会在 Player 不可用的时候将 Player 替代为一个 PlayerProxy。网络失败时，PlayerProxy 对象仍可以显示玩家的名字和最近的状态，而且在原 Player 对象无法起作用时，PlayerProxy 对象仍能继续正常工作。因此，代理类隐藏了底层 Player 类的一些不受欢迎的语义。

### 26.3.2 实现代理

Player 类的公共接口如下。sendInstantMessage() 方法需要得到网络连通性才能正常工作。

```
class Player
{
    public:
        virtual string getName();

        // Sends an instant message to the player over the network
        // and returns the reply as a string. Network connectivity
        // is required.
        virtual string sendInstantMessage(const string& inMessage) const;
};
```

代理类通常会导致 is-a 和 has-a 之争。可以把 PlayerProxy 实现为一个完全独立的类，其中包含一个 Player 对象。如果程序只要想与 Player 对象会话就会使用 PlayerProxy，这种设计就最有意义。或者，可以把 PlayerProxy 实现为一个子类，它要覆盖需要网络连通性的功能。基于这种设计，可以在失去网络连通性的时候，很轻松地把 Player 换为 PlayerProxy。这个例子使用了后一种方法，即派生 Player 的子类，如下所示：

```
class PlayerProxy : public Player
{
public:
    virtual string sendInstantMessage(const string& inMessage) const;
};
```

PlayerProxy 中 sendInstantMessage () 方法的实现只是去掉了网络功能，并返回一个字符串指示已经下线的玩家。

```
string PlayerProxy::sendInstantMessage(const string& inMessage)
{
    return "The player could not be contacted.";
}
```

### 26.3.3 使用代理

如果代理编写得当，使用代理与使用其他对象不应有什么不同。对于 PlayerProxy 示例，使用代理的代码可以完全不知道这个代理的存在。我们专门设计了以下函数，在 Player 胜利时会调用这个函数，它能处理实际的 Player，也可以处理一个 PlayerProxy。这段代码能以同样的方式处理这两种情况，因为代码能够保证有合法的结果。

```
bool informWinner(const Player* inPlayer)
{
    string result;

    result = inPlayer->sendInstantMessage("You have won! Want to play again?");

    if (result == "yes") {
        cout << inPlayer->getName() << " wants to play again" << endl;
        return true;
    } else {
        // The player said no, or is offline.
        cout << inPlayer->getName() << " does not want to play again" << endl;
        return false;
    }
}
```

## 26.4 适配器模式

之所以要修改一个类给定的抽象，原因并不一定是需要隐藏功能或出于性能方面的考虑。有时，底层抽象不能变，不过它不满足当前设计。在这种情况下，可以建立一个适配器（adapter）或包装器（wrapper）类。适配器提供了余下代码使用的抽象，相当于所需抽象与实际底层代码之间的一座桥梁。你已经看到了 STL 就使用了适配器。还记得，STL 提供了容器适配器，如 stack 和 queue，它们与其他容器（如 deque 和 list）的包装器。

### 26.4.1 举例：适配一个 XML 库

在第 24 章中，我们谈到了 Xerces XML 解析库。Xerces 是一个很不错的通用工具，它实现了许多很复杂的 XML 标准，并且提供了相当的灵活性。不过，出于某些原因，你可能需要在 Xerces 之外有一个包装器。你的用例可能很简单，只需要 Xerces 功能的一个子集。通过编写一个包装器，就可以尽可能容



易地使用有关的特性。另外，在 Xerces 之上加一个包装器还能使你有充分的自由，可以在不同 XML 库之间切换。你可能预见到以后的趋势是朝着定制 XML 代码走，或者要让用户编写他们自己的 XML 解析代码。只要代码所支持的接口与包装器的接口相同，就能正常工作。

## 26.4.2 适配器的实现

编写适配器的第一步是阅读并理解要适配的类或库。如果你对 Xerces 还不熟悉，在继续之前应该回顾一下第 24 章的内容。下一步是定义底层功能的新接口。对于这个例子，可以假设用户只需要第 24 章讨论的 Xerces 特性，即能够读取 XML 元素、属性和文本节点。可以用一个类 `ParsedXMLElement` 作为 Xerces 的适配器。客户从文件创建一个 `ParsedXMLElement`，它表示根节点。该元素的所有子元素也表示为 `ParsedXMLElement`。以下类定义显示了 `ParsedXMLElement` 的公共功能：

```
// ParsedXMLElement.h

#include <string>
#include <vector>

class ParsedXMLElement
{
public:
    ParsedXMLElement(const std::string& inFilename);
    ~ParsedXMLElement();

    std::string getName() const;
    std::string getTextData() const;
    std::string getAttributeValue(const std::string& inKey) const;
    std::vector<ParsedXMLElement*> getSubElements() const;
};
```

由于适配器会在后台使用 Xerces，所以需要为这个类定义增加一些东西。`ParsedXMLElement` 要负责在创建第一个 `ParsedXMLElement` 根对象时初始化 Xerces 库，并在最后一个根对象删除时终止库。为了实现这个功能，`ParsedXMLElement` 需要保留现有根元素对象数的一个静态计数。另外，每个 `ParsedXMLElement` 会包含 Xerces `DOMElement` 的一个指针，其中包含所解析的数据。只要相关的 `DOMElements` 存在，`XercesDOMParser` 对象就要一直存在。解析器存在于根对象中，所以有注释警告称，只要根元素撤销，子元素就无效。以下是修改后的 `ParsedXMLElement` 定义：

```
// ParsedXMLElement.h

#include <string>
#include <vector>

#include <xercesc/util/PlatformUtils.hpp>
#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/dom/DOM.hpp>
```

```
XERCES_CPP_NAMESPACE_USE
```

```
/**
 * Note: If the root element is deleted, subelements become
 * invalid.
 */
```



```

class ParsedXMLElement
{
public:
    ParsedXMLElement(const std::string& inFilename);
    ~ParsedXMLElement();

    std::string getName() const;
    std::string getTextData() const;
    std::string getAttributeValue(const std::string& inKey) const;
    // The caller is responsible for freeing the ParsedXMLElements
    // pointed to by the elements of the vector.
    std::vector<ParsedXMLElement*> getSubElements() const;

protected:
    // This constructor is used internally to create subelements.
    ParsedXMLElement(DOMElement* inElement);

    XercesDOMParser* mParser;
    DOMElement*      mElement;
    static int        sReferences;

private:
    // Disallow copy construction and op=.
    ParsedXMLElement(const ParsedXMLElement&);
    ParsedXMLElement& operator=(const ParsedXMLElement& rhs);
};

```

包装器的实现与第 24 章所示的例子很相似，所以在此不会深入细节，以下只给出了代码。重点在于，ParsedXMLElement 的每个公共方法实际上都包装着一个 Xerces 调用。我们认为 ParsedXMLElement 为 Xerces 的这个功能子集提供了一个更友好的接口，希望你也能同意这种观点：

```

#include "ParsedXMLElement.h"

#include <xercesc/util/XMLString.hpp>

#include <iostream>

XERCES_CPP_NAMESPACE_USE
using namespace std;

// No references by default
int ParsedXMLElement::sReferences = 0;

ParsedXMLElement::ParsedXMLElement(const std::string& inFilename)
{
    if (sReferences == 0) {
        // First element--initialize the library
        XMLPlatformUtils::Initialize();
    }
    sReferences++;

    mParser = new XercesDOMParser();
    mParser->parse(inFilename.c_str());

    DOMNode* node = mParser->getDocument();
}

```

```

DOMDocument* document = dynamic_cast<DOMDocument*>(node);
if (document == NULL) {
    cerr << "WARNING: No XML document!" << endl;
    return;
}

mElement = dynamic_cast<const DOMELEMENT*>(document->getDocumentElement());
if (mElement == NULL) {
    cerr << "WARNING: XML Document had no root element!" << endl;
}
}

ParsedXMLElement::~ParsedXMLElement()
{
    if (mParser != NULL) {
        // This is the root element.
        delete mParser;

        sReferences--;
        if (sReferences == 0) {
            // Last element destroyed
            XMLPlatformUtils::Terminate();
        }
    }
}

string ParsedXMLElement::getName() const
{
    char* tagName = XMLString::transcode(mElement->getTagName());
    string result(tagName);
    XMLString::release(&tagName);

    return result;
}

string ParsedXMLElement::getTextData() const
{
    // We assume that the first text node we reach is the one we want.
    DOMNodeList* children = mElement->getChildNodes();
    for (int i = 0; i < children->getLength(); i++) {
        DOMText* textNode = dynamic_cast<DOMText*>(children->item(i));
        if (textNode != NULL) {
            char* textData = XMLString::transcode(textNode->getData());
            string result(textData);
            XMLString::release(&textData);
            return result;
        }
    }

    // No text nodes were found.
    return "";
}

string ParsedXMLElement::getAttributeValue(const std::string& inKey) const
{
    XMLCh* key = XMLString::transcode(inKey.c_str());

```

```

const XMLCh* value = mElement->getAttribute(key);
XMLString::release(&key);

char* valueString = XMLString::transcode(value);
string result(valueString);
XMLString::release(&valueString);

return result;
}

vector<ParsedXMLElement*> ParsedXMLElement::getSubElements() const
{
    vector<ParsedXMLElement*> result;

    DOMNodeList* children = mElement->getChildNodes();
    for (int i = 0; i < children->getLength(); i++) {
        DOMELEMENT* elNode = dynamic_cast<DOMELEMENT*>(children->item(i));
        if (elNode != NULL) {
            result.push_back(new ParsedXMLElement(elNode));
        }
    }

    return result;
}

ParsedXMLElement::ParsedXMLElement(DOMELEMENT* inElement)
{
    mParser = NULL; // No parser for a subelement
    mElement = inElement;
}

```

### 26.4.3 使用适配器

由于适配器的存在是想要为底层功能提供一个更合适的接口，所以适配器的使用应当很直接，尤其是特殊情况。在前一个例子基础上，以下程序会输出从一个 XML 文件选出的信息：

```

int main(int argc, char** argv)
{
    ParsedXMLElement e("test.xml");
    cout << "root name: " << e.getName() << endl;

    vector<ParsedXMLElement*> subelements = e.getSubElements();
    for (vector<ParsedXMLElement*>::iterator it = subelements.begin();
         it != subelements.end(); ++it) {
        cout << "subelement name: " << (*it)->getName() << endl;
        cout << "subelement speaker: " << (*it)->getAttributeValue("speaker")
              << endl;
        cout << "subelement text data: " << (*it)->getTextData() << endl;
    }

    for (vector<ParsedXMLElement*>::iterator it = subelements.begin();
         it != subelements.end(); ++it) {
        delete *it;
    }
}

```

```
return 0;
```

针对第 24 章的例子使用上述程序时，输出如下所示：

```
root name: dialogue
subelement name: sentence
subelement speaker: Marni
subelement text data: Let's go get some ice cream.
subelement name: sentence
subelement speaker: Scott
subelement text data: After I'm done writing this C++ book.
```

## 26.5 装饰器模式

装饰器 (decorator) 模式顾名思义，就是对对象的一个“装饰”。这种模式用于在运行时改变一个对象的行为。装饰器与子类非常相似，但是其作用是临时性的。例如，如果你有一个数据流有待解析，而且到达了表示一个图像的数据，可以临时性地用一个 `ImageStream` 对象来装饰流对象。`ImageStream` 构造函数会取流对象为参数，而且已经内置有图像解析功能。图像一旦解析完毕，就可以继续使用原来的对象解析流的余下部分。`ImageStream` 相当于一个装饰器，因为它为一个现有对象（流）增加了新的功能（图像解析）。

### 26.5.1 举例：定义网页中的样式

你已经了解了，网页是以一种简单的基于文本的结构编写的，这称为超文本标记语言 (Hypertext Markup Language, HTML)。在 HTML 中，可以使用样式标记对文本应用样式，如 `<B>` 和 `</B>` 表示粗体，`<I>` 和 `</I>` 表示斜体。以下 HTML 行会用粗体显示消息。

```
<B> A party? For me? Thanks! </B>
```

下一行会用粗斜体显示消息：

```
<I> <B> A party? For me? Thanks! </B> </I>
```

假设你在编写一个 HTML 编辑应用。你的用户能够键入成段的文本，而且可以对文本应用一种或多种样式。可以为每一个样式的段落建立一个新的子类，如图 26-5 所示，不过这种设计可能很麻烦，而且随着新样式的增加会呈指数增长。

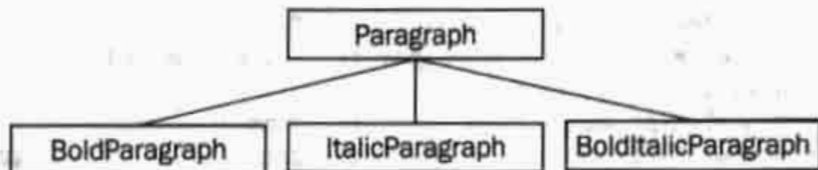


图 26-5

另一种做法是，不要把有样式的段落考虑为不同类型的段落，而要看作是加装饰的段落。这就会得到如图 26-6 所示的情况，在此 `ItalicParagraph` 作用于一个 `BoldParagraph` 之上，而 `BoldParagraph` 作用于 `Paragraph` 之上。这种对象的递归装饰会在代码中嵌套样式，就像在 HTML 中嵌套一样。

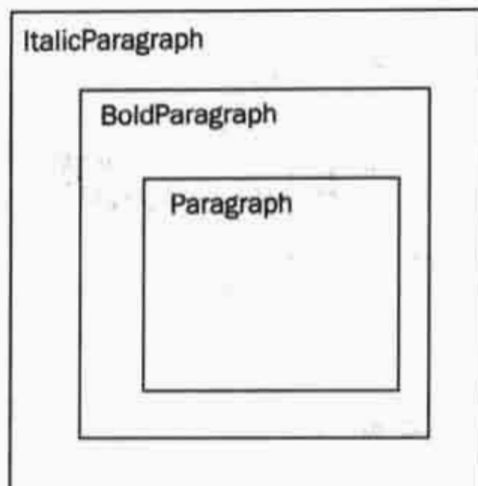


图 26-6

### 26.5.2 装饰器的实现

要用 0 或多种样式装饰 Paragraph 类，需要一个有样式 Paragraph 类的层次体系。每个有样式的 Paragraph 类都能够从一个现有的 Paragraph 构造。这样一来，它们就都能装饰一个 Paragraph 或一个有样式 Paragraph。实现有样式类的最方便的做法是派生 Paragraph 的子类。以下是 Paragraph 基类，其中提供了内联方法实现：

```

class Paragraph
{
public:
    Paragraph(const string& inInitialText) : mText(inInitialText) {}

    virtual string getHTML() const { return mText; }

protected:
    string mText;
};
  
```

BoldParagraph 类将是 Paragraph 的一个子类，这样它就能覆盖 getHTML()。不过，由于我们想把它用作为一个装饰器，惟一的公共非复制构造函数会取 Paragraph 的一个 const 引用作为参数。需要注意，它向 Paragraph 构造函数传递了一个空字符串，因为 BoldParagraph 没有使用 mText 数据成员，它派生 Paragraph 的惟一用途就是覆盖 getHTML()。

```

class BoldParagraph : public Paragraph
{
public:
    BoldParagraph(const Paragraph& inParagraph) :
        Paragraph(""), mWrapped(inParagraph) {}

    virtual string getHTML() const {
        return "<B>" + mWrapped.getHTML() + "</B>";
    }

protected:
    const Paragraph& mWrapped;
};
  
```



ItalicParagraph 类几乎是相同的：

```
class ItalicParagraph : public Paragraph
{
public:
    ItalicParagraph(const Paragraph& inParagraph) :
        Paragraph(""), mWrapped(inParagraph) {}

    virtual string getHTML() const {
        return "<I>" + mWrapped.getHTML() + "</I>";
    }

protected:
    const Paragraph& mWrapped;
};
```

重申一句，要记住 BoldParagraph 和 ItalicParagraph 派生 Paragraph 的目的只是要覆盖 getHTML()。段落的内容来自所包装的对象，而不是来自 mText 数据成员。

### 26.5.3 使用装饰器

从用户看来，装饰器模式很有意义，因为它应用起来很容易，而且一旦应用就是透明的。客户根本不必知道应用了一个装饰器。BoldParagraph 的表现就像是一个 Paragraph。

以下是一个创建和输出段落的小程序，首先用粗体显示，然后用粗斜体显示：

```
int main(int argc, char** argv)
{
    Paragraph p("A party? For me? Thanks!");

    // Bold
    cout << BoldParagraph(p).getHTML() << endl;

    // Bold and Italic
    cout << ItalicParagraph(BoldParagraph(p)).getHTML() << endl;
}
```

程序的输出如下：

```
<B>A party? For me? Thanks!</B>
<I><B>A party? For me? Thanks!</B></I>
```

这个实现有一个有意思的副作用，不过只是针对 HTML 时才会有正确的表现。如果在一行中把同一个样式应用了两次，这种效果只会出现一次：

```
cout << BoldParagraph(BoldParagraph(p)).getHTML() << endl;
```

这一行的结果是：

```
<B>A party? For me? Thanks!</B>
```

如果你能说出所以然来，说明你已经掌握了 C++！这里并没有使用带 const Paragraph 引用的 BoldParagraph 构造函数，编译器使用了 BoldParagraph 内置的复制构造函数。在 HTML 中，这是对的，毕竟没有双粗体一说。不过，使用类似框架建立的其他装饰器可能需要实现复制构造函数，来适当地设置引用。



## 26.6 职责链模式

如果想让一个面向对象层次体系中的每一个类各司其职地完成某个特定动作, 就可以使用职责链(chain of responsibility)。这种技术一般利用多态, 从而使最特定的类最先得得到调用, 它可能真正处理此调用, 也可能将调用上传到父类。父类再做同样的决定, 可以直接处理调用, 也可以继续上传到再上一级父类。职责链不一定非得遵循一个类层次体系, 不过一般都是这样。

职责链也许最常用于事件处理。许多现代应用(特别是带图形用户界面的应用)都设计为一系列事件和响应。例如, 当用户单击文件(File)菜单, 并选择打开(Open)文件时, 会发生一个打开事件。当用户在一个画图程序的可绘制区域点击鼠标时, 则出现一个鼠标按下事件。画形状时, 会一直发生鼠标移动事件, 直到最后的一个鼠标松开事件。每个操作系统都有自己的事件命名和使用方式, 不过总的思想都是一样的。事件发生时, 会以某种方式告诉程序, 程序再采取适当的动作。

可以看到, C++ 没有为图形程序设计提供任何内置的工具。它也没有事件、事件传输或事件处理的概念。对于事件处理来说, 职责链就是一个很可行的方法, 因为在一个面向对象层次体系中, 事件的处理通常与类/子类结构存在映射。

### 26.6.1 举例: 事件处理

请考虑一个绘制程序, 它有一个 Shape 类的层次体系, 如图 26-7 所示。

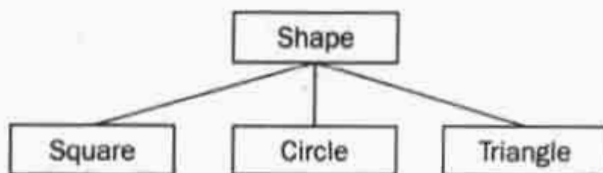


图 26-7

由叶节点处理某种特定事件。例如, Square 或 Circle 可以接收鼠标按下事件, 它会选择所选定的形状。父类会处理一些“一般性”事件, 即不论特定的形状是什么, 相应的事件处理效果是一样的。例如, 会以同样的方式处理删除事件, 而不论所删除的是什么形状。处理一个特定事件时, 最理想的算法是从叶节点开始, 在层次体系中向上遍历, 直到消息得到处理。换句话说, 如果在一个 Square 对象上发生了一个鼠标按下事件, 首先 Square 会得到机会处理此事件。如果它不认识这个事件, 其父类 Shape 则得到机会。这种方法就是职责链的一个例子, 因为每个子类可能把消息上传到链中的下一个类。

### 26.6.2 职责链的实现

取决于操作系统如何处理事件, 对消息完成这种串链的代码会有所不同, 不过与以下代码应该基本相同, 在此使用整数来表示事件的类型。

```
void Square::handleMessage(int inMessage)
{
    switch (inMessage) {
        case kMessageMouseDown:
            handleMouseDown();
            break;
        case kMessageInvert:
```

```
        handleInvert();
        break;

    default:
        // Message not recognized--chain to superclass
        Shape::handleMessage(inMessage);
    }
}

void Shape::handleMessage(int inMessage)
{
    switch (inMessage) {
        case kMessageDelete:
            handleDelete();
            break;

        default:
            cerr << "Unrecognized message received: " << inMessage << endl;
            break;
    }
}
```

程序或框架的事件处理部分接收到一个消息时，它会找到相应的形状，并调用 `handleMessage()`。通过多态，会调用子类版本的 `handleMessage()` 方法。这就让叶节点首先有机会处理此消息。如果它不知道如何处理，则会向上传至超类，超类再得到机会处理此事件。在这个例子中，消息最后的接收者无法处理事件时，只是打印一个错误。还可以抛出一个异常，或者让 `handleMessage()` 方法返回一个布尔值指示成功与否。

需要注意，尽管事件链通常与类层次体系有关，但是也不尽然。在前面的例子中，`Square` 类也可以很容易地把消息传递至一个完全不同于超类的对象。职责链方法很灵活，对于面向对象层次体系来说是一种很合适的做法。其缺点在于，要求程序员必须很认真。如果忘记从一个子类向上串链至超类，事件最后就会被丢掉。更糟糕的是，如果链到了错误的类上，可能最后会陷入一个无限循环！

### 26.6.3 使用职责链

职责链要对事件做出响应，必须有另一个类将事件分派至正确的对象。由于这个任务因框架或平台的不同有很大差异，所以下面只给出一个处理鼠标按下事件的伪代码，而不是特定于平台的 C++ 代码。

```
MouseLocation loc = getClickLocation();

Shape* clickedShape = findShapeAtLocation(loc);

clickedShape->handleMessage(kMessageMouseDown);
```

## 26.7 观察者模式

完成事件处理的另一个常用模型称为观察者（observer）、监听者消息传递（listener messaging）或发布订购（publish and subscribe）。通常更常使用这种模型，因为相对于消息链来说，这种模型不那么容易出错。采用发布订购技术，各个对象会向一个中心事件处理注册库注册它们理解的事件。接收到一个事件时，则传输至已订购的一组对象。

### 26.7.1 举例：事件处理

与前面的职责链模式类似，观察者也通常用于处理事件。这两种模式的主要区别在于，职责链最适用于逻辑层次体系，即需要找出处理事件的适当的类。观察者则适用于另一种情况，即事件可以由多个对象处理，或者与层次体系无关。

### 26.7.2 实现观察者

下面的例子中给出了一个简单的事件注册类的定义。它允许任何扩展了混合类 Listener 的对象都能订购一个或多个事件。在此还包含一个方法，在接收到一个事件时，程序就会调用此方法，它会把事件分发给所有已订购的 Listener。

```
/**
 * Listener.h
 *
 * Mix-in class for objects that are able to respond to events.
 */

class Listener
{
public:
    virtual void handleMessage(int inMessage) const = 0;
};

/**
 * EventRegistry.h
 *
 * Maintains a directory of Listeners and their corresponding events. Also
 * handles transmission of an event to the appropriate Listener.
 */

#include "Listener.h"
#include <vector>
#include <map>

class EventRegistry
{
public:
    static void registerListener(int inMessage, const Listener* inListener);

    static void handleMessage(int inMessage);

protected:
    static std::map<int, std::vector<const Listener*>> sListenerMap;
};
```

以下是 EventRegistry 类的定义。注册一个新 Listener 时，这个 Listener 会增加到一个 Listener 引用向量中，这个向量存储在相应事件的监听者映射中。接收到一个事件时，注册库只是获得这个向量，并把事件传递给向量中的每个 Listener。

```
/**
 * EventRegistry.cpp
 *
```

```

* Implements the EventRegistry class
*/

#include "EventRegistry.h"
#include <iostream>

using namespace std;
// Define the static map.
map<int, vector<const Listener*> > EventRegistry::sListenerMap;

void EventRegistry::registerListener(int inMessage, const Listener* inListener)
{
    // Recall from Chapter 21 that indexing into a map adds the element
    // with the specified key if it does not already exist.
    sListenerMap[inMessage].push_back(inListener);
}

void EventRegistry::handleMessage(int inMessage)
{
    // Check to see if the message has "any" listeners.
    if (sListenerMap.find(inMessage) == sListenerMap.end()) return;

    for (int i = 0; i < sListenerMap[inMessage].size(); i++) {
        sListenerMap[inMessage].at(i)->handleMessage(inMessage);
    }
}

```

### 26.7.3 使用观察者

下面是一个非常简单的单元测试，展示了如何使用发布订阅技术。类 `TestListener` 在其构造函数中订阅了消息 0。在构造函数中订阅消息，这对于作为 `Listener` 的对象来说是一种常用的模式。这个类中包括两个标志，用以记录是否成功地接收到消息 0，以及是否接收到某些未知的消息。如果接收到消息 0，而且没有接收到未知的消息，测试则通过。

```

class TestListener : public Listener
{
public:
    TestListener();

    void handleMessage(int inMessage);

    bool fMessage0Received;
    bool fUnknownMessageReceived;
};

TestListener::TestListener()
{
    fMessage0Received = false;
    fUnknownMessageReceived = false;

    // Subscribe to event 0.
    EventRegistry::registerListener(0, this);
}

```

```
void TestListener::handleMessage(int inMessage)
{
    switch (inMessage) {
        case 0:
            fMessage0Received = true;
            break;

        default:
            fUnknownMessageReceived = true;
            break;
    }
}

int main(int argc, char** argv)
{
    TestListener tl;

    EventRegistry::handleMessage(0);
    EventRegistry::handleMessage(1);
    EventRegistry::handleMessage(2);

    if (!tl.fMessage0Received) {
        cout << "TEST FAILED: Message 0 was not received" << endl;
    } else if (tl.fUnknownMessageReceived) {
        cout << "TEST FAILED: TestListener received unknown message" << endl;
    } else {
        cout << "TEST PASSED" << endl;
    }
}
```

取决于环境提供的服务以及自己的需求，你的程序中的具体实现可能与这里所示的实现有所不同。你可能已经注意到了，这个实现没有提供对 Listener 取消注册的方法。除非所有对象都能保证永远严格遵守规定，否则为了避免 bug，很有必要在删除对象时取消注册。这个实现还允许对象注册两次，对于你的用例来说这可能是不应该的。

## 26.8 小结

模式有助于组织面向对象概念，从而建立高级设计，这一章首先让你对此有所认识。在 Portland Pattern Repository Wiki ([www.c2.com](http://www.c2.com)) 上提供了不计其数的设计模式名录及相关讨论。如果想找到一个适用于你的任务的特定模式，很容易在众多模式中迷失方向。建议把重点放在你感兴趣的几个模式上，而且要强调学习如何开发模式，而不只是分析类似模式之间的微小差别。援引一句老话“教给我一个设计模式，我能用它编写一天的程序。教我如何创建设计模式，将让我一生的编程获益”。

设计模式作为专业 C++ 编程之旅的最后一程非常合适，因为设计模式是很好的例子，可以展示出好的 C++ 程序员如何成为最棒的 C++ 程序员。在设计中充分考虑，在面向对象编程过程中尝试多种不同方法，向你的代码库中有选择地增加新技术，你就能把自己的 C++ 技能提高到专业水平。



# 附录

## 附录 A C++ 面试宝典

阅读这本书肯定能让你的 C++ 生涯有一个爆发式的跃进，不过老板们在花钱雇用你之前，肯定希望你能证明自己。不同公司的面试方法不同，但是对于技术面试这一关，很多方面还是有章可循的。力求全面的面试人员可能想对你的编码技能、调试技能、设计和风格方面的技能，以及问题解决技能统统都测试一番。可能会问到的问题有很多。在这个附录里，你将了解到可能会遇到的一些不同类型的问题，在此还提供了一些最佳策略可以帮助你获得一份高薪的 C++ 编程工作。

这个附录还是按照本书章节的顺序来组织，我们会讨论在面试情况下，针对各章的内容哪些方面会被问到。每一节都会讨论面试人员可能会设计哪些问题来测试你有关的技能，并指出可以采用哪些最佳方法来对付这些问题。

### A.1 第 1 章：C++ 快速入门

技术面试人员经常会准备一些基本的 C++ 问题，如果有人只是听说过 C++ 而已，就在简历中写上“有 C++ 经验”想要滥竽充数，这些问题就能验明他们的正身，而无法蒙混过关。这种问题可能在“电话场景”中就会问到，在叫你参加现场面试之前先由一个开发人员或招聘人员打电话给你。也可能是通过电子邮件或当面询问。在回答这些问题时，要记住，面试人员只是想知道你确实学过并用过 C++。一般不需要为获得高分过于深入到每个细节中。

#### A.1.1 要点

- `main()` 及其参数
- 头文件语法，而且要知道标准库头文件名中没有“.h”
- 命名空间的基本使用
- 语言基本知识，如循环语法、三元操作符和变量
- 栈和堆之间的差别
- 动态分配的数组
- `const` 的使用

#### A.1.2 常见问题

基本 C++ 问题一般会采用术语测试的形式提出。面试人员可能让你定义一些 C++ 术语，如 `const` 或



static。你只要回答出课本上的答案就可以让面试人员满意了，不过，如果你再给出一个使用示例或提供额外的一些细节，可能会得到加分。例如，除了说明 const 的一个作用是可以指定引用参数不能被修改，还可以指出在向函数或方法传递一个对象时，const 引用要比复制对象效率更高。

基本 C++ 问题还可能以另一种形式出现，也许让你当着面试人员的面编写一个小程序。他会给你一个“热身”问题，如“用 C++ 写 *Hello, World* 程序”。如果拿到这样一个看来很简单的问题，你要保证能充分展现自己的技能来获得所有加分，如可以使用命名空间，使用流而不是 `printf()`，而且要知道应该包含哪些标准头文件。

## A.2 第2章：设计专业的 C++ 程序

面试人员可能想确定你除了知道 C++ 语言外，对如何应用这种语言也很精通。在此并不是明确地问你一个设计问题，不过好的面试人员会把许多技术“暗藏”在其他一些问题中，后面就会看到。

### A.2.1 要点

- 设计是主观的，要准备好在面试中为你的设计决定“辩护”。
- 在面试之前，回忆一下以前所做设计的一些细节，以备面试时可能会要求你提供一个例子。
- 准备好可能要定义抽象 (abstraction) 这个术语，并提供一个例子。
- 准备好可能会让你指出代码重用有哪些好处。
- 准备好要以可视化的方式画出一个设计，包括类层次体系。

### A.2.2 常见问题

面试人员一般很难提问设计问题，在面试情况下无论编写多大的程序，也不足以展示实际的设计技能。设计问题可能会以一种比较模糊的方式提出“告诉我设计一个好的程序有哪几步？”或者“请解释抽象的原则”。还可能更隐含一些，在讨论你先前的工作时，面试人员可能会说“你能解释一下那个项目的设计吗？”。

## A.3 第3章：基于对象的设计

面向对象设计问题可以把 C 程序员从 C++ 程序员中筛出来，C 程序员只知道引用是什么，而 C++ 程序员确实会使用 C++ 语言的面向对象特性。面试人员不会想当然地相信你，即使你使用面向对象语言已经很多年了，他们还是希望看到真正的证据来表明你确实理解面向对象方法。

### A.3.1 要点

- 过程式和面向对象模式的区别
- 类和对象的区别
- 用组件、属性和行为来表示类
- is-a 和 has-a 关系
- 多重继承时的折衷

### A.3.2 常见问题

面向对象设计问题一般有两种问法。可能让你定义一个面向对象概念，或者可能让你画出一个面向对象层次体系。前者相当简单。要记住提供例子可以给你加分。

如果让你画出一个面向对象层次体系，面试人员通常会提供一个简单的应用，如扑克牌游戏，你应

该能为这个游戏设计一个类层次体系。面试人员经常会问有关游戏的问题，因为这是大多数人熟悉的应用。与数据库实现之类的问题相比，这些游戏问题还可以稍微活跃一下气氛。当然，取决于所问到的游戏或应用，你生成的层次体系可能有很大差别。以下是要考虑的一些要点：

- 面试人员想了解你的思考过程。要把想法充分表现出来，进行头脑风暴，与面试人员一同讨论，不要怕和面试人员的观点不一致。
- 面试人员可能认为你对所问的应用已经很熟悉。如果你从来没有听说过 21 点扑克牌游戏，可以指出来，让面试人员把问题说清楚一些，或者换个问题。
- 除非面试人员要求你在描述层次体系时使用一种特定的格式，否则，建议你的类图采用继承树的形式，并为每个类提供一个大致的方法和数据成员列表。
- 你可能要为你的设计“辩护”，或者把新增加的需求考虑在内，对设计进行修改。

要搞清楚面试人员到底是什么目的，是想看你的设计中是否存在问题，还是就是要故意与你有不同观点，来看你的说服力如何。

## A.4 第4章：基于库和模式的设计

你的准老板想知道你能不能使用并不是你写的代码。如果你在简历中罗列了一些特定的库，就应该准备好回答有关的问题。如果没有，对库的重要性有一个一般的理解就足够了。

### A.4.1 要点

- 从头开始构建和重用既有代码之间的折衷
- 大 O 记法的基础知识（或者，至少要记住  $O(n \log n)$  要优于  $O(n^2)$ ）（译者注：这种说法只适用于问题规模大的情况）
- C++ 标准库中包含的功能
- 设计模式的高层定义

### A.4.2 常见问题

如果面试人员问有关一个特定库的内容，他（她）可能会强调库的高层方面而不是技术细节。例如，本书作者之一就经常问应聘者这样一个问题，从库设计的角度看，STL 的优点和缺点是什么。最好的应聘者会指出 STL 的广度和标准化是它的强大之处，但它的学习曲线很陡，这是 STL 的主要缺点。

你可能会被问到一个设计问题，但是如果这个问题有关于一个库，可能听上去并不像是一个设计问题。例如，面试人员可能会问你如何创建一个应用从网上下载 MP3 音乐，并在本地计算机上播放。这个问题并没有明确与库相关，但是这正是关键所在：这个问题实际上问的是过程。应该先说明你将如何收集需求，如何建立初始原型。由于这个问题提到了两个特定的技术，面试人员可能想了解你会如何处理这些技术。在此库就要登场了。如果告诉面试人员你要编写自己的 Web 类和播放 MP3 的代码，并不一定导致你面试失败，但是也许会让你衡量一下从头开始建立这样一些工具需要多少时间，代价如何，这就会把你难住。更好的回答是指出你会调查现有的库，看看有没有完成 Web 和 MP3 功能的库可以满足项目要求。还可以提到一些你了解的技术，如 Linux 中可以采用 libcurl 完成 Web 获取，或者在 Windows 中可以使用 Windows Media 库来完成音乐回放。

## A.5 第5章：重用设计

面试人员很少会问到设计可重用代码的问题。疏忽了这一点会带来不好的后果，如果只会写一次性代码的程序员通过了面试，得到机会参与团队开发，他们往往会对整个编程组织带来危害。有时，你可

能发现某个公司特别强调代码重用，在面试时会问到这方面的问题。如果真的问到了重用方面的问题，这就表明这是一家不错的公司，值得你为它效力！

### A.5.1 要点

- 抽象原则
- 创建子系统和类层次体系
- 好的接口设计的一般原则
- 什么时候使用模板，什么时候使用继承

### A.5.2 常见问题

有关重用的问题大多都有关于你先前开发过的项目。例如，如果你曾经为某家公司工作过，该公司同时推出两套视频编辑应用，分别面向普通消费者和专业人员，面试人员可能就会问这两个应用之间如何共享代码。即使没有明确地问你有关代码重用的问题，你也能“隐蔽”地谈到。在介绍你以前的工作时，可以告诉面试人员你写的模块是否用于其他项目中。即使是回答相当直接的编码问题，也一定要考虑并提到所涉及的接口。

## A.6 第6章：充分利用软件工程方法

如果你顺利地通过了一家公司的整个面试，但是面试人员从头到尾都没有问到任何过程问题，你就有所怀疑了，这可能说明这家公司根本没有过程，或者他们不关心过程。还有一种可能，也许他们不希望被你他们的庞大过程吓跑。大多数情况下，你都有机会问公司一些问题。建议你考虑把工程过程作为标准问题之一。

### A.6.1 要点

- 传统的生命期模型
- 形式化模型（如统一开发过程）的权衡考虑
- 极限编程的主要原则
- 你用过的其他过程

### A.6.2 常见问题

最常问到的问题就是要求你介绍一下以前的老板所用的过程。在回答这个问题时，应该指出哪些地方成功了，哪些地方失败了，不过不要对任何特定的方法本身妄加断言。你批评的方法没准正是你的面试人员所用的方法。如果你看不上极限编程，现在先不要表露出来。

本书作者曾经花大量的时间翻看应聘人员的简历，有一个趋势很明显，这就是如今每一个人都把极限编程列为他们的一项技能。尽管在这方面没有多少确凿的数据可以证明，但是在编程环境中严格遵循极限编程往往不太可能。我们了解的情况是，许多公司已经开始涉入极限编程，而且采纳了极限编程的一些原则，但是并没有正式地真正实施。

如果面试人员问你极限编程的问题，他或她可能不希望你只是背书上的定义，面试人员知道，你可能看过一本极限编程书的目录。应该挑出极限编程中你觉得有意义的一些思想。向面试人员解释这些思想，同时要加上自己的理解。尽量和面试人员展开讨论，根据面试人员言辞里的蛛丝马迹，让讨论朝着他（她）感兴趣的方向走。

你与面试人员讨论极限编程之美高级概念的时间越多，他纠缠一些细节问题（如模板语法）的时间就越少。当然你肯定不能完全躲过技术问题，不过可以借此尽量减少！

## A.7 第7章：好的编码风格

有机会参与专业编程的人都应该见过，有的同事的代码编写得相当简明。如果一个人编写的代码一塌糊涂，肯定没有人愿意同他共事。所以面试人员有时想明确应聘人员在编码风格方面的技能。

### A.7.1 要点

- 风格问题，即使面试问题并不是明确地与风格相关！
- 写得好的代码不需要太多注释。
- 注释可以传达一些元信息。
- 分解的原则
- 重构的原则
- 命名技术

### A.7.2 常见问题

风格问题会以几种不同的形式提出。本书作者之一就曾经遇到过这样一个问题，要求他在白板上写出一个相当复杂的算法的代码。他写出第一个变量名时，面试人员就让他停下来，告诉他已经通过了。问题并不在于算法，真正意图是想了解他能怎样对变量命名。更常见的情况是，可能会让你提交一份以前写的代码，或者只是让你说说对风格的观点。

如果你的准老板让你提交代码时，可要注意。把你为以前老板编写的代码提交出去可能并不合法。而且你必须找出一段合适的代码，既能展现你的技能，又不需要太多背景知识。例如，如果要面试一份数据库管理员的职位，肯定不会把有关高速图像渲染的硕士论文提交给面试公司。

如果面试公司让你写一个特定的程序，这是一个极好的机会，可以充分展现你从这本书里学到了什么。其他应聘人员可能不会为他们的程序提供单元测试，或者加上充分的注释。即使准老板没有指定程序，你也应该考虑专门编写一个小程序提交给公司。不用选择你以前编写的一些代码，可以从头开始编写与所应聘工作相关的代码，而且要强调好的风格。

## A.8 第8、9章：类和对象

关于类和对象，可能问到的问题门类相当繁多。有些面试人员比较看注语法，可能会给你看（或者让你编写）一些复杂的代码。另外一些可能不太关心实现，而对你的设计技能更感兴趣。

### A.8.1 要点

- 基本类定义语法
- 方法和数据成员的访问限定符
- this 指针的使用
- 对象创建和撤销
- 编译器在哪些情况下会为你生成构造函数
- 初始化列表

- 复制构造函数和赋值操作符
- mutable 关键字
- 方法重载和默认参数
- 友元类

## A.8.2 常见问题

像“关键字 mutable 有什么含义?”之类的问题经常会在电话中问到。招聘人员可能备有一组 C++ 术语,根据回答的正确与否来确定应聘人员能否进入下一轮面试。也许无法知道所有可能问到的术语,不过要记住,其他应聘人员也将遇到同样的问题,这也是招聘人员对你们做出评判的标准之一。

面试人员和授课老师都经常会问一种查找 bug 的问题。可能会给你一段不太复杂的代码,要求指出这段代码中存在的问题。面试人员会想方设法地分析应聘人员的能力,而这种查找 bug 的问题就是可能用到的一种方法。一般来讲,应当仔细阅读每一行代码,把你的想法说出来,充分地进行头脑风暴。bug 可能有以下几类:

- 语法错误。这种错误比较少见,面试人员知道你可以利用编译器找出编译时 bug。
- 内存错误。这包括内存泄漏和双重删除等问题。
- “不能做”的问题。这种错误包括理论上讲正确但是结果不合理的一些问题。
- 风格错误。即使面试人员不会说这是一个 bug,但还是应该指出不好的注释或变量名。

比如要找出以下程序的 bug,这里就表现出上述几个方面的问题。

```
class Buggy
{
    Buggy(int param);
    ~Buggy();

    double fjord(double inVal);
    int fjord(double inVal);

protected:
    void turtle(int i = 7, int j);
    int param;
    double* graphicDimension;
};

Buggy::Buggy(int param)
{
    param = param;
    graphicDimension = new double;
}

Buggy::~~Buggy()
{
}

double Buggy::fjord(double inVal)
{
    return inVal * param;
}

int Buggy::fjord(double inVal)
{
}
```

```

    return (int)fjord(inVal);
}

void Buggy::turtle(int i, int j)
{
    cout << "i is " << i << ", j is " << j << endl;
}

```

请仔细查看代码，然后参考以下正确的版本，找出答案。

```

#include <iostream> // Streams are used in the implementation.

class Buggy
{
public: // These should probably be public or else the class is pretty useless.
    Buggy(int inParam); // Parameter naming
    ~Buggy();

    Buggy(const Buggy& src); // Provide copy ctor and operator=
    Buggy& operator=(const Buggy& rhs); // when the class has dynamically
    // allocated memory.

    double fjord(double inVal); // int version won't compile
    // (overloaded methods differ only
    // in return type). It's also useless
    // because it just returns the argument
    // it's given.

protected:
    void turtle(int i, int j); // Only last arguments can have defaults.
    int mParam; // Data member naming
    double* graphicDimension;
};

Buggy::Buggy(int inParam) : mParam(inParam) // Avoid name ambiguity.
{
    graphicDimension = new double;
}

Buggy::~Buggy()
{
    delete graphicDimension; // Avoid memory leak.
}

Buggy::Buggy(const Buggy& src)
{
    graphicDimension = new double;
    *graphicDimension = *(src.graphicDimension);
}

Buggy& Buggy::operator=(const Buggy& rhs)
{
    if (this == &rhs) {
        return (*this);
    }
}

```



```
delete graphicDimension;  
graphicDimension = new double;  
*graphicDimension = *(rhs.graphicDimension);  
return (*this);  
}  
  
double Buggy::fjord(double inVal)  
{  
    return inVal * mParam;    // Changed data member name  
}  
  
void Buggy::turtle(int i, int j)  
{  
    std::cout << "i is " << i << ", j is " << j << std::endl; // Namespaces  
}
```

## A.9 第 10 章：探索继承技术

有关继承的问题一般与类问题形式相同。面试人员可能还要求你实现一个类层次体系来表明你确实用过 C++，而且用得不少，能够派生子类，而不仅仅只是书面了解。

### A.9.1 要点

- 派生一个类的语法
- 从子类的角度看，私有（private）和保护（protected）的区别
- 方法覆盖和虚方法
- 串链构造函数
- 向上和向下类型强制转换的输入和输出
- 多态原则
- 纯虚方法和抽象基类
- 多重继承
- 运行时类型识别

### A.9.2 常见问题

继承问题中的许多陷阱都与细节有关。在编写一个基类时，不要忘记给方法加上关键字 virtual。如果把所有方法都加上 virtual，要准备好应该能解释这种做法。要能解释 virtual 的含义及其原理。类似地，在子类定义中，不要忘记在父类名的前面加上 public 关键字（例如 class Foo : public Bar）。一般不会在面试时让你完成多重继承。

更有难度的继承问题与超类和子类的关系有关。一定要明白不同访问级别如何工作，以及私有和保护之间的区别。要提醒自己一种称为切割的现象，即某些类型的强制转换会导致一个类丢失其子类信息。

## A.10 第 11 章：利用模板编写通用代码

作为 C++ 中最神秘的一部分，面试人员会充分利用模板把 C++ 新手从高手中区分出来。如果你没有记住某些高级模板语法，尽管大多数面试人员都会原谅你，但是你起码要展示出你了解基本的模板语法。

### A. 10.1 要点

- 如何编写一个基本的模板类
- 模板的两个主要缺点，语法难看和代码膨胀
- 如何使用模板类

### A. 10.2 常见问题

许多面试问题都会从一个简单的问题入手，然后逐步增加复杂度。通常，面试人员准备的问题复杂性可以无限延伸，他只是想知道你能够应付到哪一级。比方说，面试人员开始时可能要求你创建一个类，为固定数目的 `int` 提供顺序访问。接下来，可能要求这个类能够扩展以适应一个任意大小的数组。然后，可能需要有任意的数据类型（而不只是 `int`），在这里模板可以上场了。在此之后，面试人员可以让问题朝着不同的方向发展，比如让你使用操作符重载来提供数组型的语法，或者继续沿着模板的道路让你提供一个默认类型。

一般不会明确地问模板问题，模板更有可能在解决其他编码问题时用到。你应该好好复习有关的基础知识，以备出现这种问题。不过，大多数面试人员都知道模板语法很难，在面试时要求写出复杂的模板代码有些过于苛刻。

## A. 11 第 12 章：理解 C++ 疑难问题

许多面试人员可能想强调一些更难解的情况，这样一来，有经验的 C++ 程序员就能充分发挥，表现出他们连 C++ 这些不寻常的领域也能够征服。有时面试人员想问一些有意思的问题，但是可能有困难，最后只好问他们能想到的最难解的问题。

### A. 11.1 要点

- 引用在声明时必须与变量绑定，而且这种绑定不能改变。
- `const` 的多种使用
- `static` 的多种使用
- C++ 中不同类型的强制转换

### A. 11.2 常见问题

让应聘人员定义 `const` 和 `static`，这是一个经典的 C++ 面试问题。这两个关键字能够作为标尺，面试人员可以用它来衡量应聘者的水平。比如，一个一般的应聘者可能会谈到 `static` 方法和 `static` 数据成员。一个好的应聘者会给出 `static` 方法和 `static` 数据成员的好例子。最棒的应聘者还知道 `static` 链接和函数中的 `static` 变量。

这一章介绍的边缘情况也常常出现在查找 bug 之类的问题中。要当心引用的误用。例如，假设有一个包含引用作为数据成员类。

```
class Gwenyth
{
    public:
        int& mCaversham;
};
```

由于 `mCaversham` 是一个引用，在类构造时它就必须与一个变量绑定。为此，需要使用一个初始化

列表。类可以取所引用的变量作为构造函数的一个参数。

```
class Gwentyth
{
    public:
        Gwentyth(int i);
        int& mCaversham;
};
```

```
Gwentyth::Gwentyth(int i) : mCaversham(i)
{
}
}
```

## A.12 第 13 章：有效的内存管理

低级程序员或有 C 背景的 C++ 程序员可能会问到与内存有关的问题。目的是想确定你是否过于重视 C++ 的面向对象方面，而忽略了底层实现细节。你可以把内存管理问题作为一个机会，来证明你确实知道底层的具体工作。

### A.12.1 要点

- 画出栈和堆将有助于你理解底层的具体工作。
- 使用 `new` 和 `delete` 而不是 `malloc()` 和 `free()`。
- 对数组使用 `new[]` 和 `delete[]`。
- 如果有一个对象指针数组，还需要为各个单个指针分配内存和释放内存，数组分配语法并没有考虑指针的情况。
- 或者，你总能这么说“当然，在实际中，我可以具体运行来找出问题。”

### A.12.2 常见问题

在查找 bug 的问题中通常包含内存问题，如双重删除、`new/new[]` 不匹配和内存泄漏等。在查看大量使用指针和数组的代码时，可以在分析每行代码时画出并更新内存的状态。即使你能马上看到答案，这样做也能让面试人员知道，你确实能够画出内存的状态。

要看应聘人员是否理解内存，还有一个好办法，就是让他回答指针和数组有什么区别。在这个时候，你心里对这二者之间的差别可能没有一个清晰的界定，这会让你有所迟疑。如果是这样，请再略读第 13 章有关的讨论。

## A.13 第 14 章：揭开 C++ I/O 的神秘面纱

如果你应聘一份编写 GUI 应用的工作，可能不会问你太多有关 I/O 流的问题，因为 GUI 应用会使用其他的机制来完成 I/O。不过，流可能会在其他问题中出现，而且，作为 C++ 的一个标准部分，只要面试人员关心，肯定就会问到这方面的问题。

### A.13.1 要点

- 流的定义
- 使用流的基本输入和输出
- 管理器的概念

- 流的类型（控制台、文件、字符串等等）
- 错误处理技术
- 国际化的重要性

### A. 13.2 常见问题

I/O 可以在任何问题中出现。比如说，面试人员可能让你读入一个包含测验分数的文件，再把读入的分数放在一个 `vector` 中。这个问题可以测试你的基本 C++ 技能以及对基本 STL 和基本 I/O 的理解。即使 I/O 只是所处理问题的一个小部分，也要仔细检查是否存在错误。如果没有很好地检查 I/O 错误，本来很好的程序也会让面试人员有些微辞。

你的面试人员可能不会专门问有关国际化的问题，不过在面试时如果使用 `wchar_t` 而不是 `char`，就能表现出你考虑到了国际化情况。如果确实要让你说说对国际化有什么经验，一定要提到从一开始就考虑国际化的重要性，还要表明你了解 C++ 的本地化工具。

## A. 14 第 15 章：处理错误

管理层有时不太会让刚刚毕业的研究生或新手承担一份重要的（高薪）工作，因为一般认为他们写不出成品质量的代码。你可以在面试过程中展示你的错误处理技能，从而向面试人员证明你不是泛泛之辈。

### A. 14.1 要点

- 将异常捕获为 `const` 引用。
- 对于成品代码，异常体系更可取，而不只是几个通用异常而已。
- C++ 中的抛出列表与 Java 中的抛出列表有所不同。
- 在抛出异常时智能指针有助于避免内存泄漏。

### A. 14.2 常见问题

你可能会拿到一个与异常直接有关的问题，除非会问得更为特定，如让你描述栈展开是如何进行的。不过，面试人员可能会特别留意你如何报告和处理错误。

当然，并不是所有程序员都理解或赞同异常。有些程序员出于性能原因甚至反对使用异常。如果面试人员让你完成某个工作时不要用异常，你就必须“还原”为传统的 `NULL` 检查和错误码。这是一个好机会，可以让你展示自己了解 `nothrow new`！

## A. 15 第 16 章：重载 C++ 操作符

在面试时，有可能不只是让你完成一个简单的操作符重载，而是要完成某个更难的工作，尽管这种情况比较少见。有些面试人员可能会准备一个高级问题，他们并不指望每个人都能正确地做出回答。操作符重载的复杂性使之完全可以作为这样的高级问题，因为很少有程序员在不检查的情况下一次就把语法写对。这说明，面试前应该好好看看这个方面。

### A. 15.1 要点

- 重载流操作符，因为这是最常见的重载操作符，而且概念上是特有的。
- 函数对象是什么，如何创建函数对象。
- 在方法操作符和全局友元函数之间如何选择。

- 有些操作符可以用其他操作符表示（也就是说，`operator<=` 可以写作对 `operator>` 的结果取反）。

### A. 15.2 常见问题

要接受一个现实：操作符重载问题（除了简单的操作符重载）确实很苛刻。问这种问题的人都知道这一点，所以如果你回答正确，肯定会对你印象加深。要想预计到具体会问你什么问题，这往往不太可能，但是操作符是有限的。只要你看过每个可重载操作符的重载例子，你就会有很好的表现。

除了让你实现一个重载操作符，还可能让你回答有关重载操作符的高级问题。查找 bug 的问题可能包含有一个重载操作符，但是这个操作符重载为完成某种不合适的行为，即并不是这个操作符从概念上讲本应完成的行为。除了语法外，还要牢记操作符重载的用例和理论。

## A. 16 第 17 章：编写高效的 C++ 程序

效率问题在面试中很常见，因为许多组织都遭遇到代码的可扩展性问题，特别需要在性能方面精通的程序员。

### A. 16.1 要点

- 语言级效率很重要，但是这方面的改进是有限的。从最后看来，设计级选择更为重要。
- 引用参数效率更高，因为引用参数可以避免复制。
- 对象池有助于避免创建和撤销对象的开销。
- 测评非常重要，可以确定哪些操作确实占用了大部分的运行时间。

### A. 16.2 常见问题

通常，面试人员会使用他们自己的产品作为例子来提问效率问题。有时面试人员会描述一个原来的设计，并指出他遇到的一些与性能有关的症状，要求应聘人员提出一个新的设计来解决这个问题。遗憾的是，面试人员可能已经解决了这个问题，而你的解决方案与他的解决方案完全相同的可能性微乎其微，这正是此类问题的一个不合适的地方。因为可能性很小，所以更要仔细检查你的设计。也许你提出的不是面试人员的解决方案，但是你的答案也许也是正确的，没准比他的新设计更好。

还可能有其他类型的效率问题，比如让你调整某些 C++ 代码来改善性能，或者描述某个算法。例如，面试人员可能给你看一段代码，其中包含太多复制或低效的循环。

## A. 17 第 18 章：开发跨平台和跨语言的应用

程序员提交的简历中都很少只列一种语言或一种技术，而且大型应用也很少只利用一种语言或技术。即使你只是应聘一份有关 C++ 的工作，面试人员也可能会问到有关其他语言的问题，特别是其他语言与 C++ 的关系。

### A. 17.1 要点

- 平台在哪些方面有区别（如体系结构、大小等等）
- 编程和写脚本之间的界线
- C++ 和其他语言的交互

## A. 17.2 常见问题

最常见的跨语言问题是对两种不同语言进行比较。你不要对某种语言发表溢美或贬低之辞，即使你确实不喜欢 Java。面试人员只是想知道你能够了解到不同语言之间的权衡，并能做出选择。

跨平台问题常常在讨论以前工作的时候问到。如果你的简历中称你曾经在一个定制的硬件平台上编写过 C++ 应用，就要准备好谈一谈你用的编译器，并指出使用那个平台时有哪些难题。

## A. 18 第 19 章：熟练地测试

你的准老板会对高超的测试能力大加赞赏。因为你的简历可能无法展示测试技能，除非你有 QA 经验，否则面试时就会遇到有关测试的问题。

### A. 18.1 要点

- 黑盒测试和白盒测试的区别
- 单元测试的概念，以及编写代码时同时编写测试
- 更高级测试技术
- 你以前工作的测试和 QA 环境：哪些可取，哪些不可取？

### A. 18.2 常见问题

面试人员可能要求你在面试时编写一些测试，不过在面试时写出的测试程序往往不太可能有足够的深度，不能算是有趣的测试。更有可能让你回答高级测试问题。要准备好介绍你的上一个工作中测试是如何完成的，还要指出你觉得其中哪些可取，哪些不可取。等你回答完之后，也可以问问面试人员的看法，这是一个很好的问题。在理想情况下，你们将就此展开对测试的讨论，还能让你对将来的工作环境有一个更好的了解。

## A. 19 第 20 章：征服调试

工程组织所需要的应聘人员是这样的，他们不仅能调试自己的代码，还能调试他们以前从未见过的代码。技术面试人员通常想衡量你的调试能力究竟如何。

### A. 19.1 要点

- 调试不是在 bug 出现时才开始，你应当提前在代码中做准备，以便出现 bug 时能够从容应对。
- 日志和调试工具是最好的帮手。
- bug 的症状可能表现得与实际的根本原因毫不相干。
- 内存图表有助于完成调试，特别是在面试过程中更需要画出内存图表。

### A. 19.2 常见问题

在面试时，你可能会被一个很复杂的调试问题难住。要记住，过程是最重要的，面试人员也可能知道这一点。即使你在面试过程中没有找出 bug，但是要让面试人员知道，你想通过哪些步骤找出问题所在。如果面试人员给你一个函数，指出它在运行时出问题了，如果应聘人员正确地说出了查找 bug 的一系列步骤，即使他没有马上找出 bug，面试人员也会给他同样的加分。



## A.20 第 21、22、23 章：标准模板库

可以看到，STL 可能很难使用。面试人员一般不会指望你背出 STL 类的细节，除非你声称自己是一个 STL 专家。如果你知道你所面试的工作会大量使用 STL，就需要在面试前一天写一些 STL 代码来恢复记忆。否则，复习 STL 的高级设计就应该足够了。

### A.20.1 要点

- 不同类型的容器及其与迭代器的关系
- vector 的基本用法，这可能是最常用的 STL 类
- 关联容器的用法，如 map
- STL 算法和一些内置算法的用途
- 如何扩展 STL（往往不太需要细节）
- 你自己对 STL 的看法

### A.20.2 常见问题

如果面试人员特别热衷于问详细的 STL 问题，能够问到什么确实没有止境。如果你对语法不太肯定，可以在面试时表明“当然在实际中，我会在《C++ 高级编程》里查，不过肯定它会这样工作……”。至少这会让面试人员感觉到，只要你基本思想正确，细节上有出入是可以原谅的。

有关 STL 的高级问题通常用来衡量你用过多多少 STL，而不会让你回忆所有细节。例如，一般的用户可能很熟悉关联和非关联容器。比较高级的用户能够定义一个迭代器，并说明迭代器如何用于容器。其他高级问题可能会问你有关 STL 算法的经验，以及你是否对 STL 做过定制实现。

## A.21 第 24 章：探讨分布式对象

因为分布式应用非常常见，可能会让你设计一个分布式系统，或者回答有关某种特定分布式技术的问题。

### A.21.1 要点

- 使用分布式计算的原因
- 分布式和网络式计算的区别
- 串行化和 RPC 的概念
- 如果你声称通晓 CORBA 或 XML，就必须有所准备

### A.21.2 常见问题

许多技术简历中都充斥着大量的缩写词。如果在你的简历中列了一项诸如 XML 的技术，你的准老板并不能知道你的水平如何。除非你具体指出“基本 XML 技能”或“XML 专家”，针对你对自己的评价，可能会问你一些问题来确定你的评价是否属实。具体地，对 XML 来说，可能会让你定义诸如模式（schema）等术语，或者要求给出一个应用于给定 XML 文档的模式。

由于 XML 如此流行，本书作者之一就曾经在面试时为应聘者提供了一个简单的 XML 文档。要求应聘人员指出所有属性、所有元素，以及所有文本节点。这让应聘者有些为难，不过这能有效地证明这个人是否用过 XML，或者只是知道它是一种类 HTML 的语法。

## A.22 第25章：结合技术和框架

第25章介绍的每种技术都能作为一个很好的面试问题。在此不再重复这一章中的内容，建议你在面试前再回过头来把第25章通读一遍，确保自己理解了每一项技术。

## A.23 第26章：应用设计模式

由于设计模式在专业领域越来越流行（许多应聘人员甚至把这列为他们的一项技能），所以你很有可能遇到一位面试人员要求你解释某种模式，给出某个模式的用例，或者实现一个模式。

### A.23.1 要点

- 模式的基本思想是作为一个可重用的面向对象设计概念
- 你在本书中读到的模式以及在工作中用到的其他模式
- 要知道成百上千的模式名字往往有冲突，所以你和面试人员可能会使用不同的词表述同一个东西。

### A.23.2 常见问题

回答有关设计模式的问题通常就像在公园散步，除非面试人员希望你了解每一种已知模式的细节。幸运的是，大多数欣赏设计模式的程序员都只是想与你讨论，了解你的观点而已。毕竟，从书上或网上查找有关概念而不是自己死记硬背，这本身也是一个好模式！

## 附录 B 参考书目

本附录提供了与 C++ 相关主题有关的一些书和在线资源，其中有的曾作为我们编写本书的参考，还有一些书和资源则建议你自己补充阅读，以便更为深入地了解有关内容或背景。

### B.1 C++

#### B.1.1 C++ 入门

- Harvey M. Deitel and Paul J. Deitel, *C++ How to Program (Fourth Edition)*, Prentice Hall, 2002, ISBN: 0-130-38474-7

一般就称之为“Deitel”书，阅读这本书不要求有任何编程经验。

- Bruce Eckel, *Thinking in C++, Volume 1: Introduction to Standard C++ (Second Edition)*, Prentice Hall, 2000, ISBN: 0-139-79809-9.

这是一本介绍 C++ 编程的绝好的书，要求读者已经了解 C。这本书可以免费从 [www.bruceeckel.com](http://www.bruceeckel.com) 得到。

- Stanley B. Lippman and Josée Lajoie, *C++ Primer (Third Edition)*, Addison Wesley, 1998, ISBN: 0-201-82470-1.

这本书不要求读者了解 C++，但是需要有使用高级面向对象语言的经验。

- Steve Oualline, *Practical C++ Programming (Second Edition)*, O'Reilly, 2003, ISBN: 0-596-00419-2.

这是一本人门性的 C++ 书，不要求有任何编程经验。

- Walter Savitch, *Problem Solving with C++: The Object of Programming (Fourth Edition)*, Addison Wesley Longman, 2002, ISBN: 0-321-11347-0.

这本书不要求有任何编程经验，常作为入门性编程课程的教材。

#### B.1.2 C++ 综合

- Marshall Cline, *C++ FAQ LITE*, [www.parashift.com/c++-faq-lite](http://www.parashift.com/c++-faq-lite).
- Marshall Cline, Greg Lomow, and Mike Girus, *C++ FAQs (Second Edition)*, Addison Wesley, 1998, ISBN: 0-201-30983-1.

这些常见问题是从 comp.lang.c++ 新闻组收集整理的，要想快速查找有关 C++ 的某个知识点，这个资源很有用处。印刷版本比在线版本包含了更多信息，不过在线版本中的内容对于大多数专业 C++ 程序员来说应该已经足够了。

- Stephen C. Dewhurst, *C++ Gotchas*, Addison Wesley, 2003, ISBN: 0-321-12518-5.

提供了 C++ 编程的 99 项特别提示。

- Bruce Eckel and Chuck Allison, *Thinking in C++, Volume 2: Practical Programming (Second Edition)*, Prentice Hall, 2003, ISBN: 0-130-35313-2.

Eckel 书的第二卷, 这本书涵盖了一些更高级的 C++ 主题。同样地, 这本书也可以免费从 [www.bruceeckel.com](http://www.bruceeckel.com) 在线获得。

- Ray Lischner, *C++ in a Nutshell*, O'Reilly, 2003, ISBN: 0-596-00298-X.

这是一本 C++ 参考书, 涵盖了从基础知识到更高级内容等所有方方面面。

- Scott Meyers, *Effective C++ (Second Edition): 50 Specific Ways to Improve Your Programs and Designs*, Addison Wesley, 1998, ISBN: 0-201-92488-9.
- Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison Wesley, 1996, ISBN: 0-201-63371-X.

这两本书对于通常被误用或误解的一些 C++ 特性提供了很好的提示和技巧。

- Stephen Prata, *C++ Primer Plus*, Sams Publishing, 2001, ISBN: 0-672-32223-4.

这是目前内容最为详尽的 C++ 书之一。

- Bjarne Stroustrup, *The C++ Programming Language (Special Third Edition)*, Addison Wesley, 2000, ISBN: 0-201-70073-5.

这是 C++ 书的“圣经”, 由 C++ 设计者本人编写, 每一个 C++ 程序员都应该拥有这本书, 不过对 C++ 新手来说, 这本书的某些地方可能不太好懂。

- *The C++ Standard: Incorporating Technical Corrigendum No. 1*, John Wiley & Sons, 2003, ISBN: 0-470-84674-7.

这本书基本就是一个 800 页的标准。它没有解释如何使用 C++, 只是指出了有哪些形式规则。除非你确实想了解 C++ 的每一个细节, 否则我们并不推荐这本书。

- <http://groups.google.com> 上的新闻组, 包括 comp.lang.c++.moderated 和 comp.std.c++.

这些新闻组包含了大量有用的信息, 不过在这里也会出现许多不当言辞和错误信息。

- *The C++ Resources Network*: [www.cplusplus.com/](http://www.cplusplus.com/).

这个网页没有听上去那么有用。在作者写本书时, 其中的 C++ 参考部分尚在建设当中。

### B. 1.3 I/O 流

- Cameron Hughes and Tracey Hughes, *Mastering the Standard C++ Classes: An Essential Reference*, Wiley, 1999, ISBN: 0-471-328-936.

这是一本介绍如何编写定制 istream 和 ostream 类的好书。

- Cameron Hughes and Tracey Hughes, *Stream Manipulators and Iterators in C++*, Professional Technical Reference, Prentice Hall, <http://phptr.com/articles/article.asp?p=171014&seqNum=2>.

这篇很棒的文章是由《*Mastering the Standard C++ Classes*》的作者撰写的, 它澄清了用 C++ 定义定制 stream 管理器的疑难问题。

- Philip Romanik and Amy Muntz, *Applied C++: Practical Techniques for Building Better Software*, Addison Wesley, 2003, ISBN: 0-321-10894-9.

这本书除了别具匠心地综合了软件开发建议和 C++ 具体内容之外, 还很好地解释了 C++ 中为什么提供本地化和 Unicode 支持。

- Joel Spolsky, *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*, [www.joelonsoftware.com/articles/](http://www.joelonsoftware.com/articles/)

Unicode.html.

读了 Joel 的关于国际化重要性的文章后, 你可能还想看他在 *Joel on Software* 上发表的其他文章。

- Unicode, Inc., *Where is my Character?*, [www.unicode.org/standard/where](http://www.unicode.org/standard/where).

这是查找 Unicode 字符、图、表的最佳资源。

#### B.1.4 C++ 标准库

- Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison Wesley, 1999, ISBN: 0-201-37926-0.

这本书涵盖了整个标准库, 包括 I/O streams 和 strings 以及容器和算法。这是一个很好的参考文献。

- Scott Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison Wesley, 2001, 0-201-74962-9.

Meyers 本着“Effective C++”的一贯精神编写了这本书。这本书提供了使用 STL 的有针对性的提示, 但不是一本纯粹的参考书或教程。

- David R. Musser, Gillmer J. Derge, and Atul Saini, *STL Tutorial and Reference Guide (Second Edition)*, Addison Wesley, 2001, ISBN: 0-201-37923-6.

这本书与 Josuttis 的书很类似, 不过只介绍了标准库中的 STL 部分。

#### B.1.5 C++ 模板

- Herb Sutter, *Sutter's Mill: Befriending Templates*, C/C++ User's Journal, [www.cuj.com/documents/s=8244/cujcexp2101sutter/sutter.htm](http://www.cuj.com/documents/s=8244/cujcexp2101sutter/sutter.htm).

一般认为, 应当让函数模板与类携手工作, 而这篇文章是我们见过的有关这一点的最好解释。

- David Vandevoorde and Nicolai M. Josuttis, *C++ Templates: The Complete Guide*, Addison Wesley, 2002, ISBN: 0-201-73484-2.

在这本书中, 你能找到想知道的有关 C++ 模板的任何内容 (也包括不想知道的内容)。这本书要求对 C++ 的各个方面有一定的认识。

#### B.2 C

- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language (Second Edition)*, Prentice Hall, 1998, ISBN: 0-13-110362-8.

这本书也称为“K and R”, 这是有关 C 语言的一本绝好的参考书, 不过作为入门不太合适。

- Peter Prinz, Tony Crawford (Translator), Ulla Kirch-Prinz, *C Pocket Reference*, O'Reilly, 2002, ISBN: 0-596-00436-2.

这是 C 中所有内容的一个简明参考。

- Eric S. Roberts, *The Art and Science of C: A Library Based Introduction to Computer Science*, Addison Wesley, 1994, ISBN: 0-201-54322-2.

- Eric S. Roberts, *Programming Abstractions in C: A Second Course in Computer Science*, Addison Wesley, 1997, ISBN: 0-201-54541-1.

这两本书很好地介绍了如何采用好的风格编写 C 程序, 通常用作入门性编程课程的教材。

- Peter Van Der Linden, *Expert C Programming: Deep C Secrets*, Pearson Education, 1994, ISBN: 0-131-77429-8.

深入探究 C 语言, 包括其演进以及其内部工作原理。

### B.3 集成 C++ 和其他语言

- Ian F. Darwin, *Java Cookbook*, O'Reilly, 2001, ISBN: 0-596-00170-3.

这本书循序渐进地介绍了如何使用 JNI 将 Java 与其他语言集成, 其中也包括 C++。

### B.4 算法和数据结构

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms (Second Edition)*, The MIT Press, 2001, ISBN: 0-262-03293-7.

这本书是最流行的人门性算法书之一, 涵盖了所有常用的数据结构和算法。本书作者就是在上研究生时从这本书第一版开始学习算法和数据结构的。

- Donald E. Knuth, *The Art of Computer Programming Volume 1: Fundamental Algorithms (Third Edition)*, Addison Wesley, 1997, ISBN: 0-201-89683-4.
- Donald E. Knuth, *The Art of Computer Programming Volume 2: Seminumerical Algorithms (Third Edition)*, Addison Wesley, 1997, ISBN: 0-201-89684-2.
- Donald E. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching (Third Edition)*, Addison Wesley, 1998, ISBN: 0-201-89685-0.

如果你力求严谨, 那么再没有比 Knuth 的这三卷系列更好的算法和数据结构书了。如果没有在研究生阶段学习过数学或计算机科学理论, 可能无法真正掌握这三本书。

- Kyle Loudon, *Mastering Algorithms with C*, O'Reilly, 1999, ISBN: 1-565-92453-3.

这是一本比较容易掌握的数据结构和算法参考书。

### B.5 开源软件

- The Open Source Initiative ([www.opensource.org](http://www.opensource.org)).
- GNU 操作系统— Free Software Foundation ([www.gnu.org](http://www.gnu.org)).

这是两个主要开源活动的相关网页, 解释了其原则, 并提供了如何得到开源软件, 以及如何参与开源软件开发的有关信息。

- sourceforge.net ([www.sourceforge.net](http://www.sourceforge.net)).

这个网站提供了许多开源项目, 查找有用的开源软件的一个很好的资源。

### B.6 软件工程方法论

- Barry W. Boehm, TRW Defense Systems Group, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, 21 (5): 61-72, 1988.

这篇里程碑性的论文描述了软件开发随时间的状态变化, 并提出螺旋模型。

- Kent Beck, *Extreme Programming Explained: Embrace Change*, Pearson Education, 1999, ISBN: 0-201-61641-6.

有一系列书促进了极限编程成为软件开发的一种新方法, 而这正是其中的一本。

- Robert T. Futrell, Donald F. Shafer, and Linda Isabell Shafer, *Quality Software Project Management*, Pearson Education, 2003, ISBN: 0-130-91297-2.

如果你负责软件开发过程的管理, 可以拿这本书作为指南。

- Robert L. Glass, *Facts and Fallacies of Software Engineering*, Pearson Education, 2002, ISBN: 0-321-11742-5.



这本书讨论了软件开发过程的各个方面,并在此过程中指出了一些不被注意的“真理”。

- Philippe Kruchten, *Rational Unified Process: An Introduction (Second Edition)*, Addison Wesley, 2000, ISBN: 0-201-70710-1.

提供了 RUP 的一个概述,包括任务和过程。

- Edward Yourdon, *Death March (Second Edition)*, Prentice Hall, 2003, ISBN: 0-131-43635-X.

这本很有意思的书讨论了软件开发的政治性和现实性。

- Rational Unified Process from IBM, [www3.software.ibm.com/ibmdl/pub/software/rational/web/demos/viewlets/rup/runtime/index.html](http://www3.software.ibm.com/ibmdl/pub/software/rational/web/demos/viewlets/rup/runtime/index.html)

IBM 的网站包含有关 RUP 的大量信息,包括以上 URL 中提供的交互式表示。

## B.7 编程风格

- Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999, ISBN: 0-201-48567-2.

这本经典的书针对识别和改进不好的代码提出了实践做法。

- James Foxall, *Practical Standards for Microsoft Visual Basic .NET*, Microsoft Press, 2002, ISBN: 0-7356-1356-7.

这本书展示了 Microsoft Windows 编码风格的原则,但采用 Visual Basic 来介绍。

- Diomidis Spinellis, *Code Reading: The Open Source Perspective*, Addison Wesley, 2003, ISBN: 0-201-79940-5.

这本独一无二的书把编程风格反过来,要求读者学会正确地读代码来成为一个好的程序员。

- Dimitri van Heesch, *Doxygen*, [www.stack.nl/~dimitri/doxygen/index.html](http://www.stack.nl/~dimitri/doxygen/index.html).

这是一个高可配置的程序,可以从源代码和注释生成文档。

## B.8 计算机体系结构

- David A. Patterson and John L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface (Second Edition)*, Morgan Kaufman, 1997, ISBN: 1-558-60428-6.
- John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach (Third Edition)*, Morgan Kaufman, 2002, ISBN: 1-558-60596-7.

这两本书提供了大多数软件工程师需要知道的有关计算机体系结构的所有信息。

## B.9 效率

- Dov Bulka and David Mayhew, *Efficient C++: Performance Programming Techniques*, Addison Wesley, 1999, ISBN: 0-201-37950-3.

这是为数不多的专门介绍高效 C++ 编程的几本书之一。在此涵盖了语言级和设计级效率。

- GNU gprof, [www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html](http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html).

提供了有关 gprof 测评工具的信息。

- Rational Software from IBM, [www-306.ibm.com/software/rational](http://www-306.ibm.com/software/rational).

Rational Quantify 是一个非常棒的(但不是免费的)测评工具。

## B.10 测试

- Elfriede Dustin, *Effective Software Testing: 50 Specific Ways to Improve Your Testing*, Addison

Wesley, 2002, ISBN: 0-201-79429-2.

尽管这本书所面向的是质量保证专业人员, 不过所有软件工程师人员都能从软件测试过程的讨论获益。

## B. 11 调试

- Gnu DeBugger (GDB), 位于 [www.gnu.org/software/gdb/gdb.html](http://www.gnu.org/software/gdb/gdb.html).

GDB 是一个很不错的符号调试工具。

- IBM 的 Rational Software, [www-306.ibm.com/software/rational](http://www-306.ibm.com/software/rational).

Rational Purify 是一个很好的 (但不是免费的) 的内存错误调试工具。

- Valgrind, at <http://valgrind.kde.org>.

这是面向 Linux 的一个开源内存调试工具。

## B. 12 分布式对象

- Jim Farley, *Java Distributed Computing*, O'Reilly, 1998, ISBN: 1-56592-206-9.

这本书提供了以 Java 为中心的分布式计算技术。

- Ron Hipschman, *How SETI@home Works*,  
[setiathome.ssl.berkeley.edu/about\\_seti/about\\_seti\\_at\\_home\\_1.html](http://setiathome.ssl.berkeley.edu/about_seti/about_seti_at_home_1.html).

介绍了 SETI@home 项目有趣的背景, 这个项目使用分布式计算来分析来自外太空的数据。

- Sassafras Software, *General KeyServer Questions*, <http://www.sassafras.com/faq/general.html>  
提供了 KeyServer 的有关信息, 这是一个使用分布式计算来控制软件许可的应用。

### B. 12.1 CORBA

- 对象管理组织 (Object Management Group) 的 CORBA 网站位于 <http://www.corba.org>

CORBA 是对象管理组织 (Object Management Group, OMG) 的一个“产品”。这个网站包含了基本的背景信息, 并且提供了所涉及具体标准的链接。

- Michi Henning and Steve Vinoski, *Advanced CORBA Programming with C++*, Addison Wesley, 1999, ISBN: 0-201-379270-9.

对于基于 Java 的 CORBA 有许多参考书, 而介绍基于 C++ 的 CORBA 的书并不多。这本书主要强调 C++, 尽管书名中有“高级”一词, 但是 CORBA 初学者也可以轻松阅读。

### B. 12.2 XML 和 SOAP

- Ethan Cerami, *Web Services Essentials*, O'Reilly, 2002, ISBN: 0-596-00224-6.

这本书解释了新兴的 Web 服务概念, 并用 Java 提供了使用 SOAP 完成分布式计算的例子。

- Erik T. Ray, *Learning XML (Second Edition)*, O'Reilly, 2003, ISBN: 0-596-00420-6.

这是事实上的 XML 标准参考, 其中讨论了 XML 模式、XPath 和 XHTML 等相关技术。

- James Snell, Doug Tidwell, Pavel Kulchenko, *Programming Web Services with SOAP*, O'Reilly, 2001, ISBN: 0-596-00095-2.

这本书讨论了 SOAP 和相关技术, 如 UDDI 和 WSD。分别用 Java、Perl、C# 和 Visual Basic 提供了例子。

- Eric van der Vlist, *XML Schema*, O'Reilly, 2002, ISBN: 0-596-00252-1.

这本书解决了 XML 模式的一些难题, 并讨论了 XML 语言的一些细微之处。

- Altova Software xmlspy, [www.xmlspy.com](http://www.xmlspy.com).

提供了有关 Altova Software 的 xmlspy 软件包的信息。

## B.13 设计模式

- Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley, 2001, ISBN: 0 201-70431-5.

为 C++ 编程提供了一种充分利用可重用代码和模式的方法。

- Cunningham and Cunningham, *The Portland Pattern Repository*, [www.c2.com/cgi/wiki? WelcomeVisitors](http://www.c2.com/cgi/wiki?WelcomeVisitors).

你可以花上一整天浏览这个普及型网站来了解设计模式。

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995, ISBN: 0 201-63361-2.

这也称为 GoF (Gang of Four) 书 (因为有四位作者), 这本书堪称是设计模式领域的经典之作。